Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

State Channel su tecnologie blockchain: un caso di studio sui giochi multiutente

Relatore:
Chiar.mo Prof.
Stefano Ferretti

Presentata da:
Francesco Moca

Sessione II
Anno Accademico 2018-2019

Alla mia famiglia

Introduzione

A cavallo tra il 2008 e il 2009 è stato introdotta la tecnologia *blockchain* con la presentazione della prima *crittovaluta*, *Bitcoin*:
una valuta digitale per la prima volta completamente decentralizzata.
Le transazioni di questa valuta sono contenute in una blockchain: un registro digitale, gestito da una rete peer-to-peer, usabile per memorizzare dati in modo trasparente, verificabile e immutabile senza la necessità di dipendere da un server centrale.
Successivamente nel 2013 è stato presentata Ethereum, un'altra piattaforma basata su blockchain, che ha introdotto un'ulteriore componente fondamentale per lo sviluppo di applicazione completamente decentralizzate: gli *smart contract*.
Uno *smart contract* è un programma immutabile, eseguito direttamente dai nodi della rete distribuita Ethereum, invocabile attraverso l'invio di una transazione.
Tramite questo nuovo strumento è possibile usare la blockchain non solo come base di dati, ma come una vera e propria piattaforma distribuita di calcolo nella quale i nodi sono incentivati economicamente a comportarsi in modo onesto.
Tuttavia queste garanzie sono possibili solamente a discapito dell'efficienza, e attualmente il numero di transazioni al secondo che può sostenere una blockchain come Ethereum è vertiginosamente basso se comparato alle alternative centralizzate.
Una delle prime applicazioni decentralizzate che ha palesato i limiti di

scalabilità intrinsechi della blockchain è stata CryptoKitties: un videogioco incentrato sulla compravendita di gatti virtuali che in seguito ad un'ondata improvvisa di popolarità aveva congestionato completamente la rete su cui si basa la piattaforma Ethereum.
L'obiettivo di questa tesi è dare una soluzione al problema della scalabilità per applicazioni decentralizzate ludiche con due giocatori.
Abbiamo sviluppato un esempio di gioco decentralizzato basato sugli *state channel*: un protocollo peer-to-peer regolato da smart contract che consente di aggiornare uno stato di gioco condiviso tra più giocatori riducendo il più possibile il numero di interazioni dirette con la blockchain necessarie.
Solamente quando i due giocatori non concordano sull'avanzamento dello stato di gioco è necessario interrogare la blockchain che viene impiegata come giudice imparziale.
In particolare nella nostra implementazione degli state channel ci siamo concentrati sulla possibilità di generare numeri casuali, elemento fondamentale in molti giochi ma solitamente difficile da implementare all'interno di soluzioni basate su blockchain.
L'applicazione presentata è il più possibile modulare per permettere facilmente l'implementazione di nuovi giochi differenti basandosi sempre sullo stesso protocollo sviluppato.
Nel capitolo 1 di questa tesi, dopo una breve introduzione sulla tecnologia blockchain e sulle differenze tra Bitcoin ed Ethereum, ci concentriamo sui limiti di scalabilità delle applicazioni decentralizzate e sono analizzate diverse possibili soluzioni a questa problematica.
Nel capitolo 2 presentiamo le opportunità che offre la blockchain al mondo dei videogiochi, descriviamo alcuni giochi basati su blockchain esistenti, per poi evidenziare gli attuali limiti dell'integrazione della tecnologia blockchain nell'ambito ludico.
Nel capitolo 3 viene descritta l'architettura generale della nostra applicazione che si può dividere in due parti: un client web, che espone

l'interfaccia di gioco e gestisce la comunicazione peer-to-peer tra i due giocatori, e la parte relativa agli smart contract, che regola il protocollo su cui si basano gli *state channel*.
Nel capitolo 4 viene descritta in maggiore dettaglio l'implementazione della nostra applicazione, descrivendo le varie tecnologie sfruttate e concentrandoci in particolare sul funzionamento del nostro protocollo basato su state channel.
Infine nel capitolo 5 vengono presentati i risultati relativi ad esperimenti fatti con l'applicazione all'interno di reti di test pubbliche, valutandone i costi e le tempistiche e facendo un confronto con una soluzione interamente basata su blockchain.

Indice

Capitolo 1

Blockchain e Scalabilità

In questo primo capitolo presenteremo il concetto di blockchain per poi fare un breve descrizione delle caratteristiche delle due implementazioni di Blockchain principali: Bitcoin ed Ethereum.
Dopo questa prima parte introduttiva, approfondiremo sulle problematiche di scalabilità di cui le tecnologie blockchain soffrono, concentrandoci in particolare su Ethereum.

1.1 Blockchain

L'idea di valuta digitale non è così nuova, ma solo recentemente è stata attuata con successo in una modalità completamente decentralizzata.
Law et al. presentarono un'idea di contante elettronico basato sulla crittografia a chiave pubblica, ma il loro approccio dipendeva ancora dal ruolo delle banche come autorità fiduciarie centrali [1].
Dwork e Naor proposero un sistema per combattere la posta indesiderata, nel quale un utente per inviare una mail doveva fornire il risultato del calcolo di una funzione computazionalmente difficile, che possiamo vedere come uno dei primi esempi dove è stato proposto di associare un asset digitale ad una dimostrazione del proprio lavoro svolto (*proof of work*) [2].

In modo simile gli autori di *b-money* [3], *RPOW* [4] e *bit gold* [5] presentavano l'idea di utilizzare la potenza di calcolo come un asset utilizzabile con valore reale, confrontandolo ad una moneta coniata [6]. Tuttavia questi approcci avevano comunque la necessità di una autorità centrale senza la quale non riuscivano a prevenire il *double-spending*, ovvero la possibilità di spendere più di una volta la stessa moneta.

1.1.1 Bitcoin

Nel 2008, il misterioso Satoshi Nakamoto ha presentato al mondo *Bitcoin*, una soluzione ai problemi che affliggevano l'implementazione di una moneta digitale, in particolar modo risolvendo il problema del *double-spending* [7].
Nakamoto ha proposto un sistema distribuito, un *timestamp server peer-to-peer* usato per generare la prova computazionale dell'ordine delle transazioni.
L'identità dell'inventore di Bitcoin è avvolta nel mistero, si hanno notizie di lui fino alla fine del 2010, quando ha interrotto la sua partecipazione attiva allo sviluppo del progetto Bitcoin.

Crittovaluta

Una moneta digitale, comunemente chiamata *crittovaluta* può essere definita come un sistema che rispetta le seguenti condizioni [8]:

- Il sistema non dipende da un'autorità centrale, il suo stato è mantenuto attraverso un consenso distribuito.
- Il sistema definisce se possono essere create nuove quantità di moneta. Se possono essere create, vengono definite le modalità di creazione e come determinare la proprietà di questi nuovi asset.
- La proprietà di ciascuna moneta può essere dimostrata grazie a sistemi crittografici.

- Il sistema permette di creare transazioni per modificare la proprietà delle crittovalute, evitando il *double-spending*.

Transazioni

La proprietà di ciascun bitcoin è definita tramite una catena di firme digitali.
Ogni transazione è definita con un hash che rappresenta univocamente la transazione e con un insieme di input e di output.
Ogni output di una transazione può essere utilizzato una sola volta come input per una nuova transazione, il tentativo di riutilizzare due volte la stessa output è vietato dalla rete per prevenire il double-spending.
Un output di una transazione che non è stato referenziato in precedenza, viene chiamato *UTXO* (output di transazione non speso).
Una transazione può avere una molteplicità di input e di output: diversi input possono essere usati per combinare piccole quantità di bitcoin precedentemente ricevuti, mentre gli output indicano trasferimenti di valore verso altri indirizzi o verso il proprio indirizzo nel caso si voglia generare un resto del pagamento della transazione.
Un *wallet* è caratterizzato da una coppia di chiavi asimmetriche: la chiave privata è usata per firmare una transazione, mentre la chiave pubblica (dalla quale è ricavabile l'*address* o indirizzo) è usata per la verifica della transazione come mostrato nella figura 1.1 [9].
La blockchain Bitcoin può quindi essere definita come un libro mastro (*ledger*) distribuito in una rete che descrive tutte le transazioni e le proprietà degli *UTXO*.
Ogni nodo della rete peer-to-peer conserva una copia di questo ledger.
Quando un utente, o meglio un wallet, vuole inviare una certa quantita di *bitcoin* ad un altro wallet, annuncia la transazione alla rete che ha il compito di verificarne la correttezza.
Tuttavia un utente potrebbe cercare di manipolare la rete generando più di una transazione che tenta di spendere lo stesso *UTXO* verso uten-

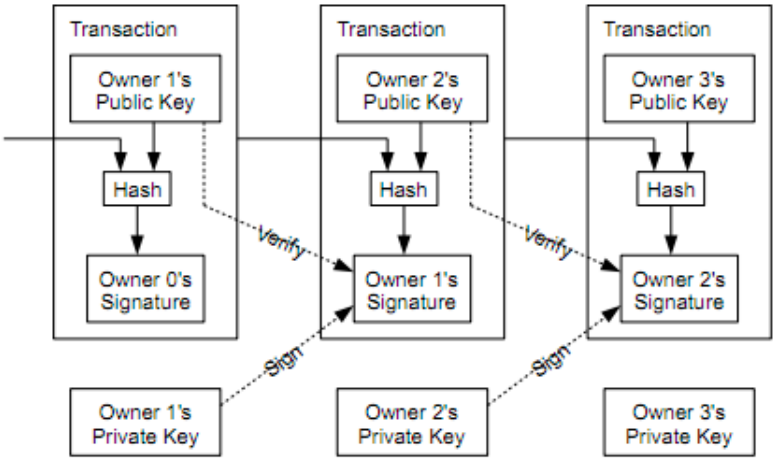


Figura 1.1: Struttura delle transazioni nella blockchain Bitcoin.

ti diversi, oppure lo stesso utente, per cercare di confermare il proprio tentativo di double-spending, potrebbe puntare a controllare molti nodi della rete Bitcoin.

Blockchain e Proof of Work

Questi tentativi di attacco possono essere prevenuti richiedendo a ciascun nodo della rete distribuita che verifica le transazioni una *proof of work* (abbreviata spesso con *PoW*).
I nodi della rete devono effettuare della computazioni difficili per provare di essere membri validi: finché la potenza computazionale totale dei nodi onesti supera quella degli attaccanti, il sistema rimarrà consistente prevenendo transazioni illecite.
Un blocco è definito come un insieme di transazioni, l'hash del blocco precedente ed il nonce usato come parametro per calcolare la *PoW*.
Un nodo detto *miner* crea un nuovo blocco indicandone l'hash e collegandolo ad un blocco annunciato precedentemente.
Attraverso questi hash è possibile definire una catena di blocchi o *bloc-*

kchain come mostrato nella figura 1.2.

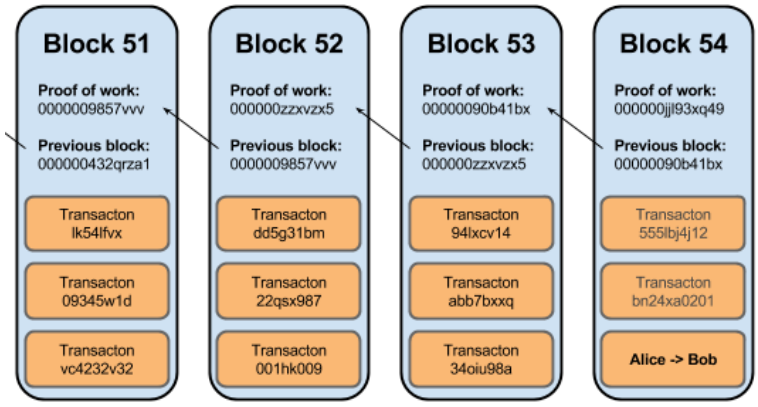


Figura 1.2: Struttura della blockchain Bitcoin.

L'algoritmo di Proof of Work usato è simile ad Hashcash [10] ed è basato sulla funzione di hashing SHA-256 [11]: il nonce all'interno del blocco è incrementato finché l'hash del blocco calcolato inizia con un numero di zeri desiderato (definito come *difficoltà* della Proof of Work). Questa operazione è definita computazionalmente difficile poichè non c'è un modo analitico per intuire il nonce corretto e quindi occorre procedere unicamente in maniera iterativa a tentativi.

Mining

La prima transazione di ogni blocco, detta *coinbase*, conia nuove monete come ricompensa per il creatore (*miner*) del blocco.
La transazione *coinbase* serve a dare un incentivo ai nodi a validare le transazioni in modo onesto, permette di mettere in circolo nuova moneta ed è una ricompensa per la dispendiosa proof of work.
Il processo di creazione di nuovi blocchi è detto mining ed avviene ad intervalli di circa 10 minuti attraverso una formula che regola la difficoltà richiesta per la proof of work successiva.
Dunque diversi utenti genereranno nuove transazioni che vengono tra-

smesse a tutti i nodi attraverso un canale di broadcast.
Ogni nodo raggruppa alcune delle transazioni non ancora verificate in un blocco e inizia a cercare il nonce che dimostri la proof of work, una volta trovato trasmette il nuovo blocco alla rete.
I nodi della rete accettano il blocco come valido solo se tutte le transazioni al suo interno sono corrette: se il nuovo blocco è accettato dalla rete, i nodi iniziano a lavorare all'aggiunta del blocco successivo.
Considerando che la rete Bitcoin è distribuita può avvenire la situazione dove più nodi annunciano quasi simultaneamente un nuovo blocco, ma con un insieme diverso di transazioni.
Questa situazione è nota come fork e porta ad un momentaneo stato incoerente della rete dove ci sono diverse catene che hanno avuto origine da blocchi diversi.
Per risolvere questa mancanza di consenso si introduce una regola che impone ad un nodo che vuole generare un nuovo blocco di dare sempre la priorità alla blockchain più lunga.
Dopo alcuni blocchi generati ci sarà un nuovo consenso tra i nodi sulla catena più lunga e le catene generate come fork non saranno valide.

Merkle Tree

Le transazioni all'interno di un blocco sono inserite in una struttura dati detta Merkle Tree [12]: un tipo di albero binario con molti nodi foglie, dove ciascun nodo è calcolato come l'hash dei relativi figli.
La figura 1.3 mostra un blocco Bitcoin composto dal Merkle tree delle transazioni, si può notare come nell'header del blocco è salvato solamente l'hash della radice dell'albero.
Tentando di alterare una qualsiasi transazione del blocco la modifica si propagherà fino all'hash della radice.
Questa proprietà è molto utile perché consente ad un nodo di risparmiare molto spazio su disco: grazie alla verifica semplificata dei pagamenti (*SPV*) i nodi possono conservare in memoria solamente la copia

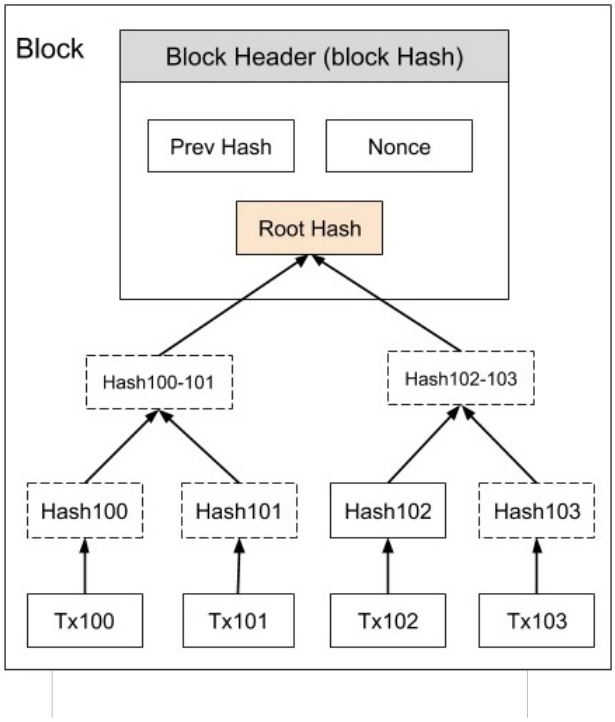


Figura 1.3: Struttura del Merkle Tree delle transazioni in un blocco.

degli header di ciascun blocco della catena più lunga anziché dover conservare il registro completo delle transazioni.

1.2 Ethereum

Ethereum è stato presentato da Vitalik Buterin nel 2014 come blockchain alternativa a Bitcoin, ottimizzata per lo sviluppo di applicazioni basate su blockchain [13].
Il principale miglioramento proposto riguarda l'introduzione di un linguaggio di programmazione che viene eseguito su una macchina virtuale che ciascun nodo della rete esegue durante la validazione delle transazioni, l'*Ethereum Virtual Machine* (EVM).
La *EVM* esegue un linguaggio bytecode a basso livello basato su stack Turing completo molto più funzionale rispetto al linguaggio di scripting

limitato che offre Bitcoin [14].
Esistono diversi linguaggi ad alto livello che possono essere usati compatibili con la EVM, il più popolare dei quali è sicuramente Solidity [15].

1.2.1 Smart Contract

L'innovazione più importante introdotta da Ethereum è la possibilità di implementare degli *smart contract* [16].
Il termine smart contract era già stato impiegato negli anni '90, dal crittografo Nick Szabo che lo definì come "un insieme di promesse, specificate in forma digitale, e i protocolli attraverso i quali i partecipanti mantengono altre promesse" [17].
Nel contesto di Ethereum, usiamo il termine smart contract per riferirci a programmi immutabili eseguiti in maniera deterministica nel contesto della EVM come parte del protocollo di rete Ethereum [18].
È importante sottolineare che il codice di un contratto prevede anche la presenza di un eventuale stato salvato in maniera permanente, chiamato *storage*.
Un'applicazione basata su smart contract è detta decentralizzata, o in breve dApp [19].

1.2.2 Account

Ethereum può esser descritto a partire dagli account, ciascuno caratterizzato da un indirizzo di 20 byte, e dal loro stato [14].
Ethereum supposta due tipi di account: gli account controllati da una chiave privata (*externally owned*), simili a quelli ad esempio di Bitcoin, gli account controllati dal codice di uno smart contract.
Ogni account è caratterizzato da 4 campi:

- **Nonce:** nel caso di account controllati da una chiave privata corrisponde al numero di transazioni inviate, mentre nel caso degli

account riferiti ad uno smart contract conta il numero di ulteriori contratti creati da esso.

- **Bilancio:** indica il numero di *wei* posseduti dall'account. Un wei indica la frazione più piccola della crittovaluta possibile, con 1 *ether* = 10^{18} Wei.

- **L'hash del codice del contratto:** campo rilevante solo per la seconda tipologia di account, rappresenta l'hash Keccak-256 del bytecode *EVM* dello smart contract.

- **Radice dello storage:** anche questo valore è rilevante solo per gli account controllati da un contratto: contiene l'hash lungo 256 bit del nodo radice dell'albero Merkle Patricia usato per rappresentare il contenuto dell'account.
  Questo *trie* è una variante ottimizzata di un Merkle Tree dove i nodi sono ordinati e le operazioni di inserimento e rimozione sono molto efficienti [20].

1.2.3 Transazioni e messaggi

Una transazione è definita con 6 campi: l'indirizzo del destinatario, la firma che indica il mittente, il valore in Ether spedito, un payload di dati opzionale, e i valori *STARTGAS* e *GASPRICE*.
Se la transazione è inviata ad un account controllato da una chiave privata, vengono semplicemente trasferiti gli ether indicati da un account all'altro.
Altrimenti se il destinatario è un contratto, può venire eseguito del codice indicato nello smart contract in risposta: considerando che il linguaggio usato dalla EVM è Turing completo un avversario potrebbe abusarne per lanciare attacchi di tipo *denial of service*.
Per prevenire che l'esecuzione di un contratto non termini mai, è stato introdotto il concetto di *gas*, unità di misura che indica un singolo passo

di computazione fondamentale.
Ogni operazione della EVM ha un costo di *gas* associato a seconda della complessità, in particolare, le operazioni che salvano dati sullo storage di un contratto sono particolarmente costose [21].
La variabile *STARTGAS* indica il massimo numero di *gas* che la transazione può consumare: se la transazione durante l'esecuzione supera il limite indicato, viene annullata senza però rimborsare il costo del gas pagato. *GASPRICE* invece indica la frazione di Ether pagata per ogni unità di *gas*, aumentare il prezzo del gas offerto diminuisce i tempi di approvazione della transazione, dato che i miner privilegiano le transazione più remunerative per loro.
Inoltre ciascun blocco ha un limite totale di gas che può contenere stabilito dai miner attraverso delle votazioni.
Una transazione dove non viene specificato il campo relativo al destinatario viene creata quando si vuole fare il deploy di un nuovo contratto nella blockchain, questa transazione restituirà l'indirizzo del nuovo account *contratto* generato.
Un contratto può generare in maniera analoga transazioni, che però sono chiamate messaggi [22].

1.3 Limiti della blockchain

Nonostante la crescente popolarità di utilizzo, una blockchain *pubblica* ha ancora diversi limiti relativi alla *scalabilità* che ne impediscono l'adozione per molti potenziali casi d'uso.
Il principale limite attuale è relativo al *throughput* delle transazioni: attualmente Ethereum può elaborare all'incirca 15 *tps* (transazioni al secondo), mentre la rete Bitcoin ne processa circa la metà [23].
Per fare un confronto con un circuito di pagamento *mainstream*, Visa in media processa 2000 tps con picchi fino a 56'000 [24].

Questo limite di prestazioni è strettamente collegato all'inefficienza di algoritmi di consenso *proof of work* e dal vincolo che ogni transazione deve esser processata da ogni singolo nodo connesso alla rete.
Questo vincolo è necessario per considerare una blockchain *authoritative*: i nodi distribuiti non si devono affidare a terze parti per rimanere aggiornati sullo stato della blockchain.
La scalabilità limitata è anche strettamente collegata alle caratteristiche di ciascun blocco: la dimensione dei blocchi e la frequenza temporale con cui vengono alla blockchain influenza ovviamente il numero di transazioni processabili nel tempo.
A parte l'utilizzo puramente monetario, blockchain programmabili come Ethereum hanno reso più interessante lo sviluppo di applicazioni decentralizzate(*dApp*).[25].
In questa tipologia di applicazioni la blockchain può rimpiazzare il ruolo di una base di dati, mentre gli *smart contract* sostituiscono il codice back end solitamente eseguito da un server centralizzato.
Questi limiti di scalabilità stanno tuttavia ostacolando lo sviluppo di questa tipologia di applicazioni.
Nel dicembre 2017, CryptoKitties, una *dApp* ludica incentrata sulla compravendita di gattini collezionabili virtuali, era diventata per un periodo molto popolare con conseguenze abbastanza preoccupanti sulle prestazioni della rete.
Le transazioni relative allo smart contract di CryptoKitties avevano superato il 10% del totale delle transazioni provocando una notevole congestione sulla rete ethereum con conseguenze temporanee sui tempi di convalida e sul costo del gas delle transazioni [26].
Come possiamo vedere dalla figura 1.4, negli anni successivi ci sono stati ulteriori incrementi improvvisi del costo delle commissioni delle transazioni dovuti a episodi simili.
Appare quindi evidente che la rete Ethereum allo stato attuale non è scalabile ed occorra trovare soluzioni a queste problematiche.

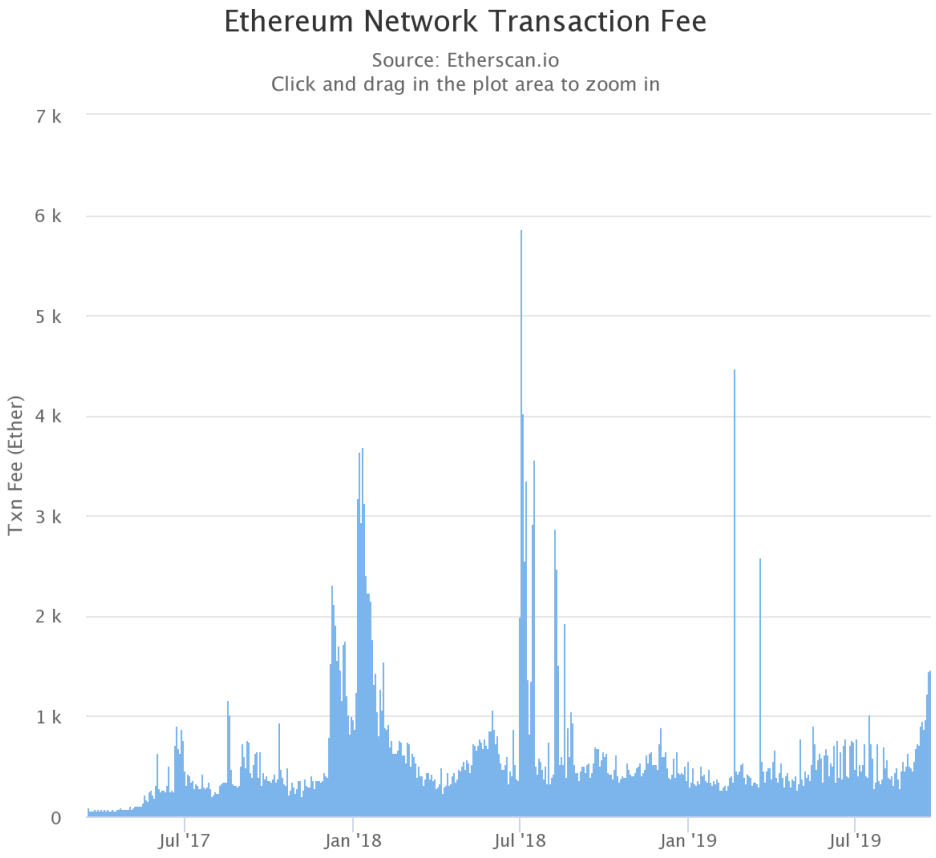


Figura 1.4: Andamento delle commissioni delle transazioni della rete Ethereum tra il 2017 e il 2019.

1.4 Il trilemma della blockchain

Analizzando i requisiti desiderabili di una blockchain si può concludere che è molto difficile soddisfarli tutti allo stesso tempo.
L'inventore di Ethereum Vitalik Buterin ha per primo introdotto un *trilemma* [27], illustrato nella figura 1.5, che afferma che una blockchain può rispettare al più due delle seguenti tre proprietà:

- *Decentralizzazione*:
  Un singolo nodo deve richiedere una quantità di risorse limita-

to ($O(c)$) per incoraggiare un grande numero di partecipanti alla rete.

- *Scalabilità*:
  La rete deve essere in grado di processare un numero di transazioni n superiori a quelle che un singolo nodo potrebbe processare nello stesso tempo: ($O(n) > O(c)$).

- *Sicurezza*:
  La rete deve resistere attacchi con potenza computazionale inferiore a quella della rete stessa.
  Nel caso di blockchain che implementano protocolli di consenso *proof of work* si considera insicuro un sistema dove il 51% della potenza computazionale è controllato da una stessa entità. [28]

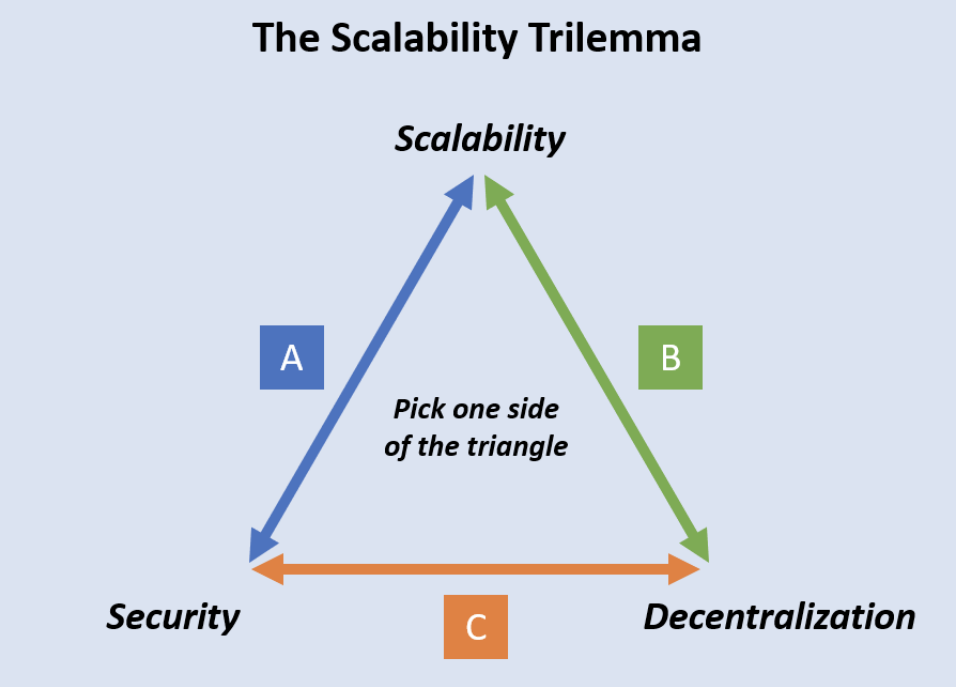


Figura 1.5: Triangolo del trilemma sulla scalabilità

Altri ricercatori hanno descritto lo stesso problema in termini simili. Trent McConaghy [29] e Greg Slepak [30] hanno introdotto il trilemma *DCS* applicabile ad un qualsiasi sistema decentralizzato.

- *Decentralizzazione*:
  Nessuna singola entità controlla la rete.
- *Consistenza o Consenso*:
  Tutti i nodi vedono gli stessi dati allo stesso tempo.
- *Scalabilità*:
  La rete deve poter scalare a livello planetario, con un throughput nell'ordine delle centinaia di migliaia di transazioni ed una latenza nell'ordine di qualche secondo.

Al momento nessuna blockchain pubblica è in grado di soddisfare tutti questi criteri contemporaneamente, ma si possono descrivere due diversi approcci che tentano di superare le limitazioni individuate da questi trilemmi:

- Cambiare il protocollo della blockchain: **Soluzioni Layer 1**
- Implementare soluzioni offchain: **Soluzioni Layer 2**

1.5 Soluzioni Layer 1

Sono le soluzioni che cercano di migliorare la scalabilità, agendo direttamente sul protocollo della blockchain.

1.5.1 Variazione caratteristiche dei blocchi

Ad esempio, è possibile modificare i parametri relativi alla dimensione dei blocchi e all'intervallo di tempo tra un blocco e l'altro.
Questi valori tuttavia se aumentati in maniera eccessiva causano una maggiore difficoltà nel raggiungere il consenso tra i nodi e una minore

decentralizzazione causata dal maggior numero di risorse computazionali richieste da ciascun nodo.
Secondo alcune simulazioni è possibile raggiungere un throughput massimo di circa 27 tps modificando i parametri relativi ai blocchi del protocollo della rete *Bitcoin* [23].
Oltre ad essere poco soddisfacente, una soluzione del genere richiederebbe il consenso della maggior parte dei miner per evitare un *hard fork* e la nascita di una nuova criptovaluta.

1.5.2 Ethereum 2.0

Interventi più efficaci implicano modifiche più radicali al protocollo come ad esempio lo *sharding* e la **proof of stake** [27].
Questi due soluzioni sono alla base della nuova versione prevista della blockchain Ethereum chiamata *Serenity* o *Ethereum 2.0* [31].

Proof of Stake

Proof of Stake (*PoS*) è una tipologia di algoritmi di consenso per blockchain pubbliche, alternativa alla *Proof of Work*, che si basano sull'interesse economico di un validatore nella rete.
Un insieme di validatori si alternano proponendo e votando il blocco successivo e il peso del voto di ciascun validatore è proporzionale alla dimensione del proprio deposito (*stake*).
In generale, in un algoritmo di Proof of Stake la blockchain tiene traccia di un insieme di validatori, e chiunque detenga una certa quantità di valuta (nel caso di Ethereum, l'ether) può diventare un validatore inviando un particolare tipo di transazione che blocca il proprio ether in un deposito [32].
Il processo di creazione e accettazione di nuovi blocchi viene poi fatto attraverso un algoritmo di consenso al quale possono partecipare tutti

i validatori attuali.
Si possono descrivere molte varianti della *PoS* e diversi modi diversi per ricompensare i validatori.
Da una prospettiva algoritmica, si possono descrivere due varianti principali [33]:

- *Proof of Stake Chain-Based*:
  Un algoritmo pseudo-casuale sceglie un validatore per un certo periodo di tempo e assegna a quel validatore il diritto di creare un singolo blocco che deve essere collegato a un qualche blocco precedente (normalmente il blocco alla fine della catena più lunga). In questo modo la maggior parte dei blocchi generati convergeranno in una singola catena di dimensione crescente.

- *Proof of Stake BFT (da Byzantine fault tolerance)*:
  Ai validatori viene assegnato casualmente il diritto di proporre i blocchi, ma per concordare se un determinato blocco vado incluso nella blockchain interviene un processo a più round dove ogni validatore invia un voto per un blocco specifico durante ogni round, e alla fine tutti i validatori (onesti e online) concordano permanentemente se un dato blocco fa parte o meno della catena.
  La differenza chiave è che il consenso su un blocco non dipende dalla lunghezza o dalle dimensioni della blockchain dopo la sua aggiunta.

Tra i vantaggi significativi della *PoS* vi sono la sicurezza, la riduzione del rischio di centralizzazione e l'efficienza energetica [34] :

- Non è necessario consumare grandi quantità di energia elettrica per garantire il consenso. Si stima che sia Bitcoin che Ethereum brucino oltre 1 milione di dollari di elettricità e hardware al giorno a causa della Proof of Work.

- Non c'è la necessità di emettere tante nuove monete per motivare i miner a continuare a partecipare alla rete.

- La *PoS* permette di applicare concetti della teoria dei giochi per scoraggiare la formazione di cartelli centralizzati che possono agire in modo dannoso per la rete (ad esempio il *selfish mining* [35])

- Riduzione dei rischi di centralizzazione che sono incoraggiati nelle blockchain basate su *PoW* dall'economia di scala.

- La possibilità di utilizzare le sanzioni economiche per rendere varie forme di attacchi del 51% molto più costosi da effettuare rispetto alla *PoW*, ad esempio distruggendo completamente il deposito degli attaccanti.

Si possono descrivere principalmente due tipologie di attacchi esclusivi a questo algoritmo di consenso [36] :

- *Nothing at stake:*
  Se c'è una biforcazione nella catena, la strategia ottimale per ogni validatore è quella di convalidare transazioni su entrambe le catene, in modo che il validatore ottenga la sua ricompensa indipendentemente da quale delle due catene si riveli poi vincente.

- *Attacco a lungo raggio*:
  Attacco simile a quello del 51% su proof of work (fare una catena più lunga che riscrive il registro a favore dell'attaccante), ma prevede di iniziare l'attacco molto più indietro nella storico della blockchain (ad esempio decine di migliaia di blocchi prima). Non essendoci nessuna prova del lavoro svolto, diventa difficile distinguere quale delle due catene sia quella legittima.

Diverse implementazioni della Proof of Stake propongono differenti soluzioni a questi attacchi.

Una particolarità dell'implementazione in lavorazione dal team di Ethereum (chiamata Casper) è che prevede un periodo di transizione dove saranno impiegate sia la Proof of Work che la Proof of Stake per poi abbandonare definitivamente la prima [37].
La complessità di questa transizione ha comportato diversi slittamenti nell'inizio di questa fase di transizione che dovrebbe iniziare ad inizio 2020 per poi concludersi negli anni seguenti [38].

Sharding

L'implementazione della Proof of Stake, permette l'adozione di tecniche di *sharding* sulla blockchain che consentiranno di migliorare notevolmente la scalabilità.
Lo *sharding* consiste nel dividere la rete in più porzioni dette "shard" o frammenti, come illustrato nella figura 1.6.
Per esempio, uno schema di *sharding* su Ethereum potrebbe mettere tutti gli indirizzi che iniziano con 0x00 in un shard, tutti gli indirizzi che iniziano con 0x01 in un altro shard e così via [39].
Nella forma più semplice di sharding, ogni shard ha anche la propria storia di transazioni e l'effetto delle transazioni in un shard è limitato ad esso.

Esiste un insieme di validatori, ai quali viene assegnato casualmente il diritto di creare blocchi di shard.
Durante ogni slot di tempo, per ogni shard viene selezionato un validatore casuale che ha il diritto di creare un blocco relativo shard.
Inoltre per ogni shard, viene selezionato un insieme di validatori detti attestatori. L'intestazione di un blocco di uno shard può essere inclusa nella "catena principale" (detta anche beacon chain), se è firmata da almeno due terzi degli attestatori.
In un sistema del genere i nodi che partecipano nella rete possono essere di diverso tipo [40] :

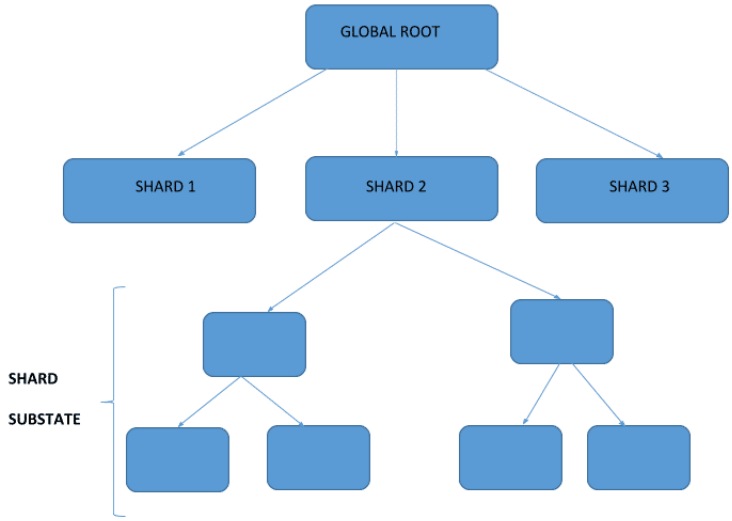


Figura 1.6: Divisione dello stato della blockchain in porzioni (Merkle Tree)

- *Nodo Super Full*:
  Scarica i dati completi della *beacon chain* e di ogni blocco riferito agli shard collegati alla beacon chain.

- *Nodo Top Level*:
  Elabora solo i blocchi della *beacon chain*, comprese le intestazioni e le firme dei blocchi degli shard, ma non scarica il contenuto di questi ultimi.

- *Nodo Single-shard*:
  Agisce come un nodo top level, ma scarica anche i dati completi riferiti a qualche shard in particolare.

- *Nodo Light*:
  Scarica e verifica solo le intestazioni di blocco dei blocchi della main chain. Non elabora nessuna transazione a meno che non

deve leggere qualche stato di un determinato shard, in quel caso scarica il ramo del Merkle Tree corrispondente a un determinato blocco di interesse.

Occorre ovviamente trovare un modo per far comunicare tra loro gli shard: nel caso Ethereum, l'approccio che si sta studiando è quello di "ricevuta" (*receipt*).
Quando una transazione all'interno di uno shard viene eseguita, può cambiare lo stato del proprio shard locale, generando anche "ricevute", che sono memorizzate in una sorta di memoria condivisa distribuita che può essere successivamente visualizzata (ma non modificata) da altri shard. Lo sharding crea una serie di potenziali nuovi vettori di attac-

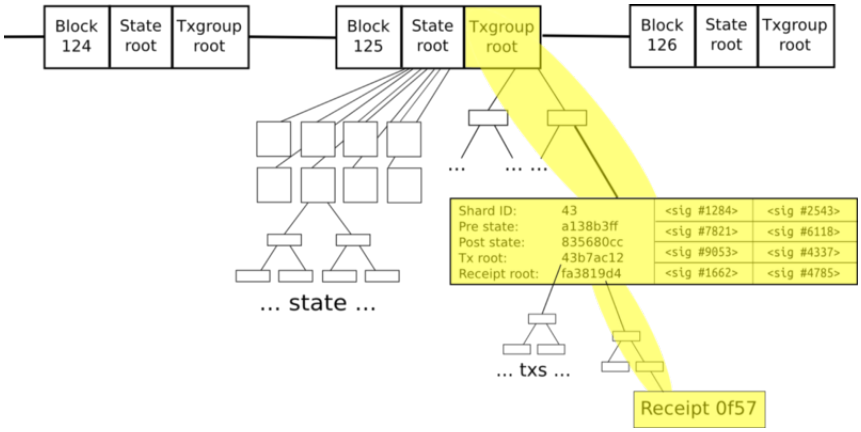


Figura 1.7: Ricevuta in un merkle tree

co come ad esempio il *single-shard takeover attack*. In questo attacco, un attaccante controlla la maggior parte dei produttori di blocchi in uno shard per creare uno shard maligno che può presentare transazioni non valide.
Gli sviluppatori di Ethereum puntano al campionamento casuale dei validatori come soluzione, ma questa soluzione è ancora in fase di sviluppo.

1.6 Soluzioni Layer 2

Altre soluzioni anziché cercare di modificare il protocollo blockchain di base, puntano a migliorare la scalabilità riducendo il numero di transazioni effettuate su blockchain, cercando comunque di mantenere la sicurezza garantita dalla blockchain.
Queste soluzioni sono implementate attraverso l'uso di smart contract, e sono più semplici da impiegare non richiedendo il consenso dei nodi della rete nel cambiare il protocollo.
Inoltre sono adattabili a diverse implementazioni di blockchain pubbliche.
Possiamo dividere le soluzioni che agiscono su un secondo livello in due categorie:

- *Sidechain e Plasma*
- *Payment Channel e State Channel*

1.6.1 Sidechain e Plasma

Un sidechain è una blockchain separata che è collegata alla blockchain principale (*parent blockchain*) tramite un collegamento bidirezionale (*two-way peg*), come illustrato nella figura 1.8.
Questo collegamento consente lo scambio degli asset ad un tasso predeterminato tra le due blockchain [41].

Un utente vincola una parte dei propri asset nella blockchain principale attraverso uno smart contract, per poi ricevere l'equivalente degli asset nella sidechain.
Analogamente in ogni momento un utente può effettuare una transazione inversa (chiamata a volte *exit*) e riconvertire i propri asset in quelli della blockchain principale
Queste sidechain solitamente hanno un thoughput molto maggiore rispetto ad una blockchain basata su proof of work poiché basano il pro-

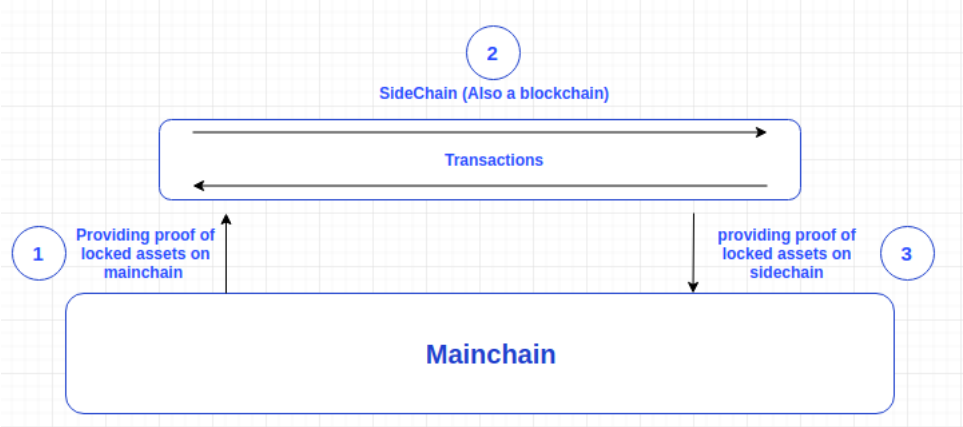


Figura 1.8: Sidechain

prio consenso su algoritmi alternativi come ad esempio *proof of stake* o *delegated proof of stake* [42].
Le *sidechain* sono vulnerabili ad un tipo di attacco chiamato Invalid State Transition. L'idea alla base dell'attacco è che la maggioranza dei validatori possono colludere per sottrarre i fondi ad altri partecipanti generando un blocco invalido e simultaneamente nello stesso blocco completare il furto effettuando la transazione di *exit*.

Plasma

Plasma è un variante di sidechain (chiamate *child chain*) basata su smart contract che ha l'obiettivo di evitare ai validatori della blockchain secondaria di potersi impadronire illecitamente degli asset vincolati nella *mainchain* [43].
Nella blockchain Plasma ogni volta che viene generato un blocco, viene inviato l'hash dell'header del nuovo blocco alla blockchain principale, questo hash viene usato per valutare eventuali tentativi di frode.
In ogni momento l'utente può decidere di uscire dalla child chain e ritirare propri i fondi nella blockchain principale, anche nel caso i validatori della child chain agiscano in modo disonesto, vedi l'esempio in

figura 1.9.

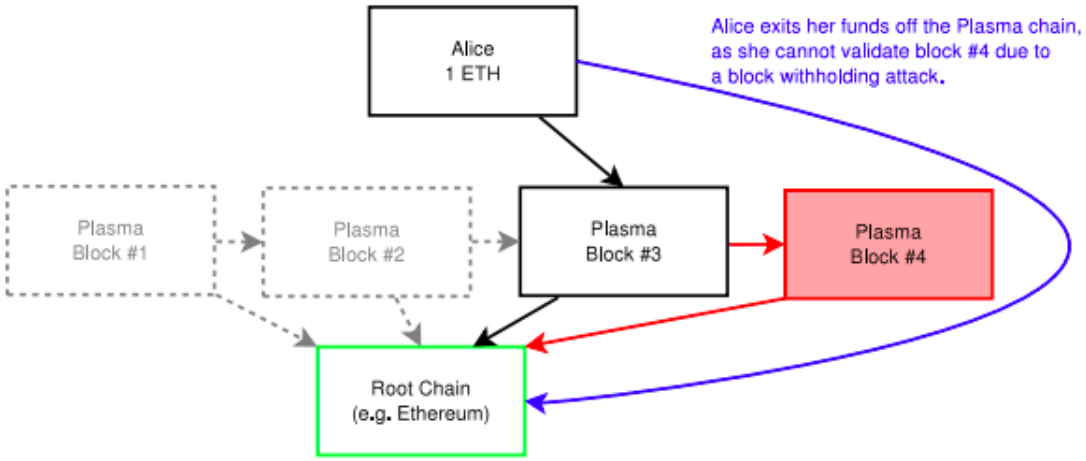


Figura 1.9: Esempio di comportamento disonesto da parte del validatore che decide di non pubblicare (*block withholding*) il nuovo blocco validato, indicato in rosso. Alice può aprire una disputa e, dopo una finestra di tempo, ritirare i fondi vincolati.

Al momento della conversione asset della child chain (che devono essere in qualche modo univoci, ad esempio *token non-fungible*) sono associati all'utente con una prova crittografica (ad esempio una chiave privata). Quando un utente vuole uscire dalla child chain deve presentare una transazione di uscita includendo una "cauzione di uscita".
Questa transazione tuttavia non è istantanea, è prevista una finestra di tempo dove ogni utente può presentare una "prova di frode" alla catena principale. La prova di frode contiene i dati di un blocco precedente che dimostra che l'uscita richiesta non è valida, in quanto i fondi che cercano di essere prelevati dall'utente sono già stati "spesi" da quell'utente in un blocco diverso.
Qualsiasi utente della side chain è interessato a dimostrare un tentati-

vo di frode poiché viene ricompensato con la cauzione di uscita versata dall'utente disonesto.

1.6.2 Payment Channel e State Channel

I *payment channel* e gli *state channel* sono protocolli usati per ridurre in numero di transazioni su Blockchain (onchain) tra utenti, preferendo quando possibile lo scambio di messaggi *offchain*.

Payment Channel

I Payment Channel sono una versione semplificata degli State Channel. Vengono creati depositando una certa quantità di fondi in uno smart contract e consentono trasferimenti bidirezionali tra due partecipanti, a condizione che la somma netta dei loro trasferimenti non superi il valore depositato.
Questi trasferimenti possono essere effettuati istantaneamente e senza alcun coinvolgimento della blockchain, a parte per l'apertura del canale e per la chiusura dello stesso.
Dopo la chiusura del canale è possibile ritirare i fondi depositati.
Viene garantita la sicurezza del trasferimento dato che ciascuna transazione è firmata da entrambi i partecipanti del canale.
È possibile combinare più payment channel creando un payment network per permettere pagamenti tra utenti che non sono direttamente collegati tra loro [44].

1.6.3 State Channel

Gli *state channel* sono una generalizzazione dei *payment channel,* dove gli utenti eseguono transazioni tra loro fuori dalla blockchain non limitandosi ai pagamenti, ma anche modificando dei dati a partire da uno stato iniziale concordato all'apertura dello state channel. [45]

Il protocollo di uno state channel tra due utenti è simile a quello descritto precedentemente per i payment channel, come illustrato nella figura 1.10:

1. Una stato iniziale, firmato da entrambi i partecipanti, è vincolato tramite una transazione *onchain* indirizzata ad uno smart contract con lo stesso effetto. Generalmente oltre allo stato iniziale entrambi i partecipanti associata un qualche asset di valore, ad esempio ether oppure token di pagamento.

2. I partecipanti, scambiandosi messaggi, aggiornano lo stato *off-chain*, firmando ogni transazione che potrebbe essere inviata alla blockchain. Ogni nuovo aggiornamento è irreversibile, ovvero rende lo stato precedente non più valido.

3. In caso di disaccordo sulla validità di un aggiornamento viene aperta una disputa risolvibile *onchain* usando uno *smart contract* su blockchain come giudice. In seguito ad una disputa si può decidere di chiudere lo state channel o continuare il protocollo.

4. Quando entrambi i partecipanti concordano su uno stato finale lo inviano alla blockchain, chiudendo lo *state channel* e sbloccando in maniera coerente i fondi pagati inizialmente.

Il punto 3 del protocollo descritto impone ai partecipanti di uno *state channel* di essere costantemente online connessi alla blockchain per poter identificare eventuali dispute e rispondere in tempo. Per ovviare a questa limitazione sono stati proposti protocolli di state channel che delegano la risoluzione delle dispute a terze parti [46]. Nel caso ottimale i partecipanti di uno state channel eseguono solamente due transazioni, rendendo ideale l'utilizzo di questo protocollo in casi d'uso dove un numero limitato di utenti ha numerose interazioni tra loro in una certa finestra temporale.
Il punto più critico relativo all'implementazione di uno state channel è

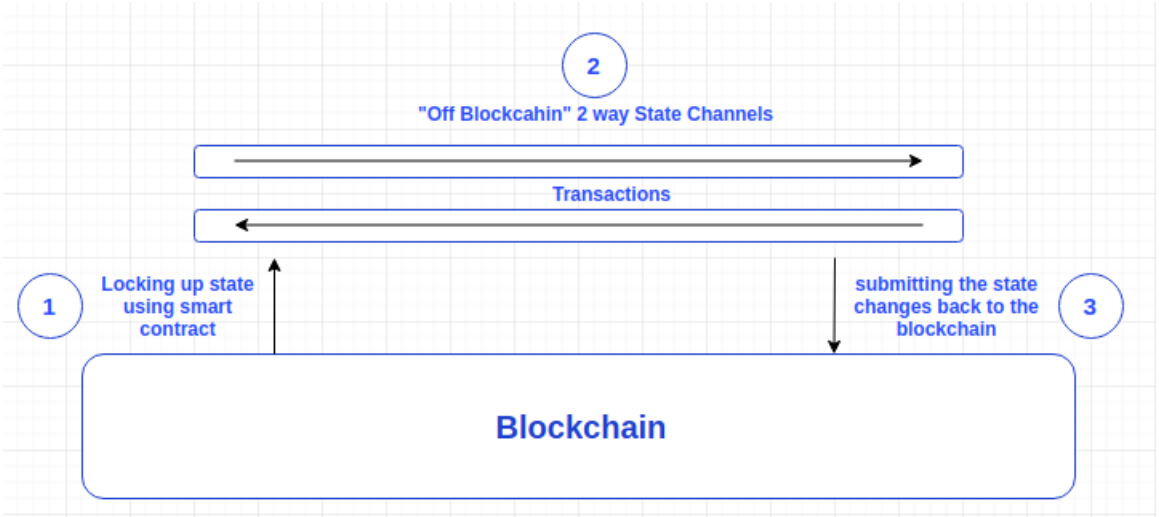


Figura 1.10: Two way State Channel

quello relativo alla risoluzione delle dispute in caso di disaccordo tra gli utenti sull'aggiornamento dello stato.
L'apertura di una disputa può essere causata dal rifiuto di uno dei due utenti al firmare un nuovo stato, l'invio di uno stato aggiornato non valido o la disconnessione di uno dei due utenti.
Un fattore ulteriore da valutare riguarda il costo economico pagato per apertura di una disputa che non dovrebbe mai superare il valore depositato dall'utente [47].
Per cercare di limitare questo problema può essere utile pensare un sistema di penalità che coinvolga gli avversari disonesti.
La maggior parte delle ricerche riguardanti gli state channel che coinvolgono due partecipanti ma sono state proposte soluzioni che creano reti di più canali permettendo la partecipazione di un numero di utenti maggiore [48].

Confronto tra Sidechain e State Channel

Entrambe le soluzioni layer 2 presentate si basano sullo stesso modello generale.

1. Uno stato e/o degli asset vengono bloccati nella blockchain.
2. Esecuzione di transazioni offchain.
3. Sblocco degli propri asset dallo state channel o dalla sidechain.

Nonostante questo ci sono molte differenze tra le due soluzioni e l'una può essere considerata peggio o meglio dell'altra a seconda del caso d'uso.
L'implementazione degli State Channel è molto più semplice, dà garanzie di privacy migliore tra i partecipanti e ciascuno aggiornamento di stato ha finalità immediata non dipendendo dalla generazione di nuovi blocchi da parte di una sidechain.
Tuttavia si rivelano adatti solamente ad applicazioni che hanno un numero di partecipanti limitato e già conosciuto inizialmente.
L'impiego di una sidechain permette di far interagire tra loro più crittovalute, ma introduce la complessità di dover interagire con due blockchain differenti nella propria applicazione.
Infine l'impiego di sidechain, anche usando soluzioni come Plasma, sono considerabili meno decentralizzate rispetto agli State Channel, dipendendo da un ulteriore rete di nodi potenzialmente disonesti che cercano di stabilire un consenso.

Capitolo 2

Gaming su blockchain

A prescindere dagli scopi finanziari, la tecnologia blockchain offre opportunità uniche in molti altri settori.
La principale novità di questa tecnologia emergente è infatti rendere possibile lo sviluppo di applicazioni distribuite e decentralizzate senza doversi fidare della buona fede dei nodi che compongono la rete peer to peer.
È possibile quindi scrivere del codice pubblico e immutabile all'interno di smart contract e avere la garanzia che questo codice venga eseguito correttamente da tutti i nodi della blockchain.
Gli effetti dell'esecuzione di codice possono causare una variazione dei dati contenuti sulla blockchain in modo trasparente e verificabile da tutti.
L'industria videoludica si adatta perfettamente alla natura dell'ecosistema delle crittovalute rendendo realizzabile il desiderio di molti giocatori: gli oggetti virtuali ottenuti giocando sono veramente di proprietà dei giocatori e sono scambiabili e persino riutilizzabili in altri giochi [49].
Tuttavia a causa la maggior parte dei giochi sviluppati su blockchain sono ancora in fase molto preliminare a causa dei problemi di scalabilità illustrati nel capitolo precedente e da altri limiti della blockchain che rendono questa tecnologia ancora troppo acerba per il mercato main-

stream.
Una volta individuate delle soluzioni a questi problemi tecnici, potrà iniziare lo sviluppo di giochi con meccaniche innovative rese possibili dall'integrazione della blockchain.

2.1 Vantaggi di giochi basati su blockchain

L'integrazione della blockchain in nuovi videogiochi può essere interessante sia dal punto di vista dei giocatori, sia da quello degli sviluppatori.

- Proprietà reale degli asset di gioco:
  Tutti gli oggetti di un gioco possono essere creati su blockchain come token fungibile (nel caso di valute di gioco) o non fungibile (nel caso di oggetti di gioco o avatar). In questo modo gli asset guadagnati giocando appartengono realmente al giocatore per sempre: lo sviluppatore non può eliminarli e il giocatore può venderli o scambiarli liberamente con altri giocatori.

- Correttezza di gioco provabile:
  Rendere la logica di gioco verificabile impedisce a giocatori disonesti di imbrogliare. Allo stesso modo rendere il codice del gioco trasparente impedisce di manipolare elementi di gioco a favore dei creatori del gioco. Ad esempio diventa impossibile manipolare le probabilità a proprio favore in un gioco d'azzardo contro il banco.

- Nuovi modelli economici:
  Molti sviluppatori di giochi su blockchain stanno proponendo nuovi modelli di gioco *play-to-earn* [50]. Questi giochi, contrapponendosi ai giochi *freemium* attualmente in tendenza, propongono un nuovo modello di business dove il profitto del gioco è condiviso ai giocatori tramite l'introduzione di incentivi economici a giocare.

- Interazione tra diversi giochi:
  Dato che i dati relativi alla proprietà degli oggetti di gioco sono chiaramente disponibili sulla blockchain, altri giochi possono usufruire di queste informazioni. È quindi possibile introdurre nuovi giochi che interagiscono con altri, incoraggiando nuove meccaniche di gioco.

- Riduzione dei costi di mantenimento:
  Realizzando un gioco decentralizzato che non si appoggia a nessun server centrale, uno sviluppatore può risparmiare sui costi del server trasferendoli sui validatori della blockchain e sugli utenti stessi.

- Partecipazione degli utenti sullo sviluppo del gioco:
  È possibile creare giochi con elementi modificabili dai giocatori stessi. Ad esempio è possibile inserire regole di gioco che i giocatori possono modificare conducendo votazioni su blockchain, o addirittura prevedere l'introduzione di nuove regole.

2.2 Panoramica storica sui giochi blockchain

Già dal 2012 è possibile risalire a giochi che si interfacciavano alla blockchain [51]. I primi giochi creati erano per lo più d'azzardo e erano ancora completamente centralizzati.
La blockchain era sfruttata unicamente come mezzo di pagamento per le vincite in gioco.
Il primo esempio di gioco ad usare effettivamente la blockchain come meccanica di gioco principale è stato *Huntercoin*, rilasciato nel 2014.
In questo gioco il client stesso è un nodo di una blockchain proprietaria. Durante il gameplay, i giocatori competono per raccogliere per primi monete virtuali (chiamati token *HUC*) che vengono fatti apparire nella mappa nel tempo.

Il gioco, inizialmente pensato come un esperimento che doveva durare solamente un anno, nel momento di maggior successo ha raggiunto decine di migliaia di giocatori unici e i token raccolti sono stati scambiati nel corso di 5 anni per un totale stimato di 1’282’417 dollari [52].
Dopo questi primi esperimenti, in tempi più recenti la maggior parte di giochi su blockchain si basa sulla piattaforma Ethereum che al momento è de facto la più popolare per la creazione di applicazioni decentralizzate che sfruttano le potenzialità degli smart contract.
Nel Dicembre 2017 proprio su questa piattaforma è diventato famoso il già citato CryptoKitties, un gioco dove utenti allevano e collezionano gatti virtuali unici rappresentati da token non fungibili.
Oltre a dimostrare i limiti attuali di scalabilità della blockchain, questo gioco ha ispirato la nascita di giochi simili che hanno approfondito l’aspetto collezionabile del gioco con altri elementi di gameplay più tradizionale.
Un altro contributo di questo gioco, è stato quello di dimostrare le potenzialità di interazioni di diversi giochi tra loro tramite la condivisione degli stessi asset di gioco. CryptoKitties infatti ora fa parte di un gruppo di giochi ("*KittyVerse*") dove è possibile usare gli stessi token che rappresentano i gatti in altri giochi sviluppati in maniera indipendente [53].

2.3 Problematiche dei giochi su blockchain

Oltre al problema principale relativo alla scalabilità trattato approfonditamente nel capitolo precedente, ci sono altre problematiche importanti che influiscono sullo sviluppo di giochi su blockchain.

2.3.1 Usabilità delle applicazioni su Blockchain

In qualsiasi applicazione, l'user experience è notevolmente influenzata dal tempo di risposta dopo un'azione da parte di un utente.
Secondo le linee guida comunemente accettate riguardanti l'usabilità, un limite accettabile di attesa per un feedback da parte di un qualsiasi sistema non dovrebbe superare un secondo.
Un utente superati i 10 secondi di attesa, perde l'attenzione su quello che stava facendo [54].
In media una singola transazione su Ethereum richiede tra i 15 secondi e i 5 minuti ad essere validata [55].
Effettuare ogni singola azione come se fosse una transazione su blockchain renderebbe l'esperienza di gioco molto deludente.

2.3.2 Generazione numeri casuali

Un ulteriore problematica riguarda la generazione di numeri casuali. Spesso i giochi necessitano di elementi di casualità, come ad esempio il lancio di un dado, ma per la natura deterministica della blockchain è complicato generare numeri casuali all'interno di una transazione.
Una soluzione *naive* a questo problema sfrutta i dati contenuti nel blocco dove è contenuta la transazione, come ad esempio l'hash dei blocchi precedenti o il timestamp della transazione, come seed per generare numeri casuali.
Tuttavia questo metodo è vulnerabile ad eventuali manipolazioni da parte dei miner, che possono decidere di pubblicare o ritardare una transazione dopo aver calcolato il numero casuale ottenuto.
Un altro approccio è quello di utilizzare un Oracolo.
Gli oracoli sono servizi di terze parti che accedono a fonti esterne, che nel nostro caso potrebbero essere servizi web che generano numeri causali, e scrivono i dati ottenuti sulla blockchain in modo che gli smart contract possano utilizzarli.

Questo approccio è ancora meno sicuro del precedente, poiché è completamente centralizzato e permetterebbe al proprietario dell'oracolo di poter segnalare un qualsiasi numero a propria scelta come casuale [56].

Soluzioni più sicure si basano su schemi crittografici *commit/reveal*, dove tutti i giocatori generano un seed casuale e pubblicano l'hash del numero generato.

Dopo che tutti i giocatori hanno pubblicato l'hash del proprio seed, viene rivelato il numero scelto. Tutti i seed vengono quindi combinati e attraverso una qualche operazione matematica viene generato un numero casuale.

Questo metodo può tuttavia rivelarsi poco efficiente poiché ogni numero generato richiede 2 transazioni(o messaggi) per ogni giocatore.

Nella soluzione implementata per la nostra applicazione, inspirata dagli State Channel per casinò descritti da FunFair [57], sarà necessario un unico messaggio per ogni numero generato.

Capitolo 3

Architettura dell'applicazione

Abbiamo sviluppato per questa tesi un esempio di soluzione per poter implementare giochi per due giocatori basati su blockchain che risolva i problemi individuati nella sezione 2.3 del capitolo precedente. Si può dividere il lavoro svolto in più parti:

1. Sviluppo di smart contract, eseguibili su blockchain Ethereum, che implementano degli state channel per due partecipanti con caratteristiche specifiche per l'implementazione di giochi su blockchain.
   In particolare si è introdotta la possibilità di usare valori casuali in modo efficiente.

2. Sviluppo di un'applicazione web *peer to peer*, che permetta la comunicazione tra due utenti, autenticati mediante la propria identità (o meglio il proprio *wallet*) su blockchain.

3. Implementazione di un semplice gioco dimostrativo, *Pig*, attraverso una semplice interfaccia web integrata alla parte descritta nel punto precedente che fa da collegamento tra il client web e la blockchain e attraverso uno smart contract che ne descriva le regole.

Si è cercato di mantenere l'applicazione il più modulare possibile per consentire con poco sforzo l'implementazione di nuovi giochi più complessi riutilizzando gli altri elementi sviluppati.

3.1 Descrizione del gioco scelto

Pig è un semplice gioco di dadi descritto per la prima volta in stampa da John Scarne nel 1945 [58].
Il gioco è spesso usato come esempio didattico per insegnare concetti di probabilità, ma è stato anche spesso impiegato in contesti informatici sia come esempio per illustrare concetti di programmazione, sia per ambiti più avanzati come il machine learning [59].
Nel nostro contesto è stato scelto poiché è un esempio di gioco con elementi casuali (il lancio del dado), lo stato di gioco è molto semplice (il punteggio dei due giocatori) ed è caratterizzato da un ordine di turno variabile.

3.1.1 Regolamento

Durante il proprio turno, un giocatore tira ripetutamente un dado a 6 facce con l'obiettivo è accumulare quanti più punti possibile, sommando i numeri ottenuti ad ogni lancio del dado.
Se un giocatore lancia un 1, il suo turno termina e tutti i punti accumulati durante il turno vengono persi.
Tirare un 1 non azzera il punteggio ottenuto nei turni precedenti, ma solo il totale guadagnato durante quel turno specifico.
Un giocatore può infine di fermarsi, somma i punti ottenuti durante il turno al proprio punteggio e passa il turno al giocatore successivo.
Quando un giocatore raggiunge almeno 100 o più punti, il gioco finisce e quel giocatore è il vincitore.

3.2 Struttura generale

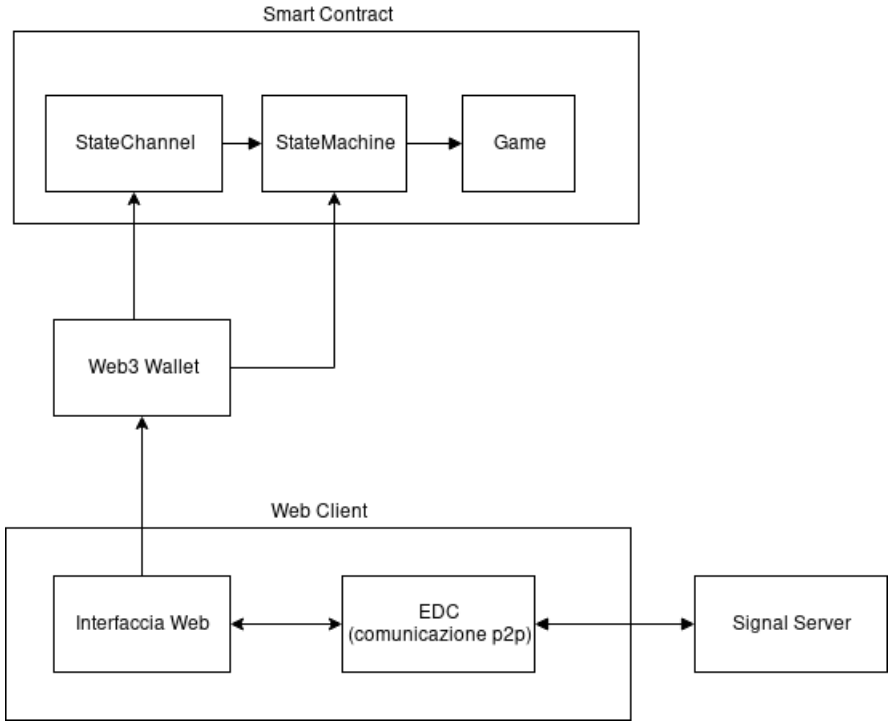


Figura 3.1: Diagramma delle componenti dell'applicazione

La figura 3.1 mostra in dettaglio le varie componenti dell'applicazione. La parte più importante del client è la libreria sviluppata **EDC** (*Ethereum Data Channel*) che permette l'apertura di un canale di comunicazione fra due client attraverso la tecnologia WebRTC.
Nella fase iniziale della connessione, il client interagisce con un server esterno per un iniziale scambio di messaggi necessario per aprire una connessione diretta tra i due client.
Dopo che la connessione è stata effettuata, il server centrale non è più impiegato e l'applicazione diventa completamente decentralizzata.
Il client web per interagire alla blockchain o per firmare i messaggi scambiati con l'altro client si interfaccia con un wallet che implementa la libreria Web3, solitamente per questo scopo è impiegata un esten-

sione browser (*Metamask*), ma esistono anche browser che offrono un wallet integrato.
Il wallet interroga un nodo Ethereum per interagire con i tre Smart Contract che compongono l'implementazione del nostro State Channel. Due diverse tipologie di interazioni avvengono tra il client e la blockchain:

- Operazioni di scrittura:
  Dati relativi allo stato dell'applicazione vengono salvati o modificati. In questo caso è necessario creare una transazione e pagare le relative commissioni.

- Operazioni di sola lettura:
  In caso si vogliano solamente leggere dati relativi allo stato dell'applicazione scritti precedentemente su blockchain non è necessario creare alcuna transazione e pagare alcuna commissione ai miner.

L'interfaccia web inizialmente si occupa di mostrare i giocatori disponibili a giocare e permette all'utente di selezionarne uno per iniziare il processo di connessione.
A connessione effettuata l'interfaccia cambia mostrando una chat per comunicare con l'altro giocatore, un riquadro che mostra messaggi di log per monitorare gli eventi relativi all'applicazione e l'interfaccia di gioco vera e propria.
Quest'ultimo componente è adattabile e sostituibile per aggiungere giochi diversi da quello implementato.

3.2.1 Struttura Smart Contract

Possiamo distinguere nell'implementazione degli smart contract, 3 parti principali.

- *State Machine*:
  Implementa un automa a stati: la funzione principale implementata del contratto prende come input una struttura che descrive uno stato ed una che descrive un'azione e ritorna un nuovo stato. Il nuovo stato è generato secondo regole deterministiche.

- *Game*:
  Definisce le regole specifiche di un singolo gioco. All'interno della State Machine viene richiamato quest'altro contratto che è l'unico tra i tre contratti che dipende dal gioco specifico.

- *State Channel*:
  Permette l'apertura e la chiusura di uno state channel. Implementa il protocollo che permette a due partecipanti di aggiornare uno stato condiviso appoggiandosi al contratto dell'automa: ciò può avvenire completamente *offchain* ma con le stesse garanzie di sicurezza come se lo stato venisse completamente aggiornato *onchain*. Gran parte del contratto si concentra su gestire le dispute tra i due partecipanti nel caso non concordino sull'avanzamento dello stato *offchain*.

3.3 Funzionamento applicazione

All'apertura dell'applicazione, e successivamente ogni tre secondi, viene inviato un messaggio in un canale di broadcast presso un server centrale pubblicizzando la propria disponibilità ad iniziare una connessione, identificandosi con il proprio indirizzo ethereum.
In questo modo in ciascun client viene popolata una lista di tutti i potenziali giocatori.
Un utente selezionando un indirizzo da quella lista, invia attraverso un canale di comunicazione privato una richiesta di connessione, firmata con la chiave privata del proprio wallet.

Il ricevente della richiesta, risponde firmando a sua volta il messaggio con la propria chiave. Se la risposta è affermativa, attraverso un ulteriore scambio di messaggi, si veda la sezione x.x relativa al protocollo WebRTC del capitolo 4 per ulteriori dettagli, viene finalmente stabilità una connessione diretta sicura.
Per iniziare una partita ad un qualsiasi gioco, i due partecipanti si accordano sullo stato iniziale (firmandolo entrambi) e sul gioco da giocare e sulla State Machine da impiegare.
Assumendo che entrambi i giocatori abbiano già depositato alcuni fondi nel canale (ad esempio 1 ether a testa), uno dei due giocatori crea una transazione verso il contratto dello State Channel usando come parametro i valori su cui entrambi si sono accordati tramite firma digitale, aprendo lo State Channel.
Una volta che la transazione è stata validata dai miner, il canale può considerarsi aperto. Per progredire lo stato del canale, l'utente sceglie un'azione e consulta offchain il contratto State Machine passandogli lo stato precedente e l'azione scelta. Se l'azione scelta è possibile, il codice del contratto restituirà un nuovo stato.
Il partecipante firma sia l'azione che il nuovo stato e li invia alla controparte che a sua volta verifica la fattibilità dello stato proposte e la validità delle firme inviate.
Se la controparte concorda con il nuovo stato proposto, firma a sua volta il nuovo stato e se vuole continuare a giocare propone a sua volta una nuova azione iniziando nuovamente il processo.
Questa sequenza di eventi si ripete fino a quando i partecipanti concordano che il canale deve essere chiuso.
A questo punto, entrambe le parti firmano una azione di chiusura che viene inviata al canale per chiuderlo.
Infine, in base al risultato della partita al gioco scelto, i due partecipanti possono ritirare i fondi che erano stati vincolati all'apertura del contratto.

Ricapitolando possiamo riassumere il protocollo descritto in questi step:

- Se è il mio turno di agire, devo farlo in un tempo ragionevole, inviando un'azione e un nuovo stato firmandoli.

- Se ricevo un nuovo stato, devo verificare che lo smart contract nella blockchain sia d'accordo con la transizione di stato proposta, verificare che sia il nuovo stato che l'azione proposta siano firmati correttamente e inviare la mia firma sul nuovo stato alla controparte.

Molte cose potrebbero però andare storte e uno dei due partecipanti potrebbe deviare dall'esecuzione standard di questo protocollo.
Ad esempio uno dei due utenti potrebbe disconnettersi, proporre uno nuovo stato errato oppure rifiutarsi di firmare lo stato corretto proposto dalla controparte.
In qualsiasi situazione anomala, uno dei due utenti può iniziare un processo di disputa *onchain*, fornendo lo stato aggiornato ritenuto corretto.
Lo Smart Contract che gestisce le dispute attende per un certo intervallo di tempo un'eventuale replica da parte dell'altro utente, stabilisce quale sia lo stato corretto, decide eventuali penalità e, a seconda della tipologia di disputa aperta, mantiene lo State Channel aperto o lo chiude.
Nel capitolo successivo, nella sezione x.x verranno descritte nel dettaglio le varie tipologie di dispute possibili.

3.3.1 Numeri Casuali

Una parte che fin'ora è stata omessa nella descrizione del funzionamento è quella relativa alla generazione dei numeri casuali all'interno dell'applicazione.
Come già descritto nel capitolo precedente la casualità è notoriamente difficile da ottenere su una blockchain poiché ogni transazione è intrin-

secamente deterministica.
Il nostro approccio utilizza un meccanismo commit-reveal:

- Entrambi i giocatori scelgono un grande numero casuale e ne calcolano l'hash.
- Gli hash vengono scambiati (non importa in quale ordine).
- Entrambi i giocatori rivelano il numero che hanno scelto. Entrambi i giocatori possono verificare che i numeri rivelati corrispondono all'hash rivelato inizialmente, e possono generare l'output casuale richiesto.
  Per farlo nella nostra implementazione, facciamo l'hash della concatenazione dei due numeri rivelati e ne calcoliamo il modulo secondo le nostre esigenze specifiche.

Questo processo viene integrato alle regole degli smart contract relativi alla State Machine e State Channel che verificano durante i vari passaggi la correttezza degli hash indicati.
Questo sistema funziona ma è piuttosto inefficiente, per ogni mossa richiederebbe 4 messaggi.
Infatti il giocatore insieme all'azione scelta dovrebbe comunicare il proprio *commit*, successivamente l'avversario risponde con il proprio commit, e in altri due messaggi i due giocatori si scambiano i relativi reveal.
Per rendere questo protocollo più efficiente implementiamo un sistema che si basa su delle catena di hash, usando numeri grandi 256 bit e keccak256 come algoritmo di hashing:

- Ogni giocatore genera un numero casuale abbastanza grande e ne calcola l'hash molte volte, ad esempio 10000, creando una lista di hash intermedi.
- Basandoci sull'assunzione che l'hash di un numero di 256 bit è sufficientemente random, possiamo generare molti numeri casuali

semplicemente considerando il *commit* del numero successivo, il *reveal* di quello precedente.

In pratica, questo significa che all'apertura del canale, ogni partecipante fornisce l'ultimo hash che ha generato come primo *commit* nel canale.
Poi associata ad ogni azione viene rivelato l'hash precedente nella lista della catena di hash.
La controparte può verificare che questo hash sia effettivamente l'inverso dell'hash rivelato precedentemente e rispondere con il proprio *reveal*.
Il protocollo per numeri casuali descritto quindi richiede solamente 2 messaggi per ogni azione che richiede un numero casuale.

Capitolo 4

Implementazione

In questo capitolo illustreremo approfonditamente i dettagli implementativi delle due parti dell'applicazione descritta nella sezione 3.2.

4.1 Applicazione Web

Il client sviluppato è una *single-page application* (*SPA*), un'applicazione web dove tutto il codice necessario, HTML, JavaScript e CSS, viene scaricato al caricamento iniziale della pagina [60].
Eventuali risorse aggiuntive sono caricate dinamicamente all'interno dell'applicazione in seguito ad azioni dell'utente, senza che la pagina web visualizzata venga aggiornata o cambiata.

4.1.1 Strumenti utilizzati

- **Metamask**: estensione per browser che permette di usare un wallet Ethereum senza la necessità di programmi esterni [61].
  Un'estensione del genere è fondamentale per lo sviluppo di una dApp poiché permette ad una pagina web di interfacciarsi direttamente alla rete *Ethereum*, senza che gli utenti debbano fornire al

sito web che stanno visitando la chiave privata del proprio indirizzo.
Di default permette di interagire alla rete *Ethereum* principale e a svariate testnet pubbliche senza dover installare alcun nodo localmente, attraverso un servizio pubblico offerto da Infura [62].

- **web3.js**: Libreria Javascript che permette di interfacciarsi ad un nodo Ethereum, nel nostro caso attraverso Metamask.
  È stata impiegata per firmare i messaggi usati per la stabilire la connessione WebRTC e successivamente per le transazioni di apertura/chiusura del canale, per la gestione delle dispute e per l'invocazione (in lettura) della State Machine.

- **simple-peer**: Libreria Javascript implementa le API WebRTC che permettono di creare canali tra due client per scambiare audio, video o nel nostro caso dati.

- **signalhub**: Server centrale usato per la fase di iniziale di connessione tra utenti. Permette ad un utente di ricevere ed inviare messaggi verso un canale di comunicazione specifico identificato da una stringa.

- **React**: Libreria Javascript sviluppata da Facebook per sviluppare *SPA*. Si basa sul concetto di Componenti che permettono di dividere un interfaccia utente in parti riusabili e indipendenti [63].

- **Redux e Rematch**: Librerie usate per gestire lo stato dell'applicazione secondo il *design pattern* Flux [64].

4.1.2 Architettura Flux per SPA

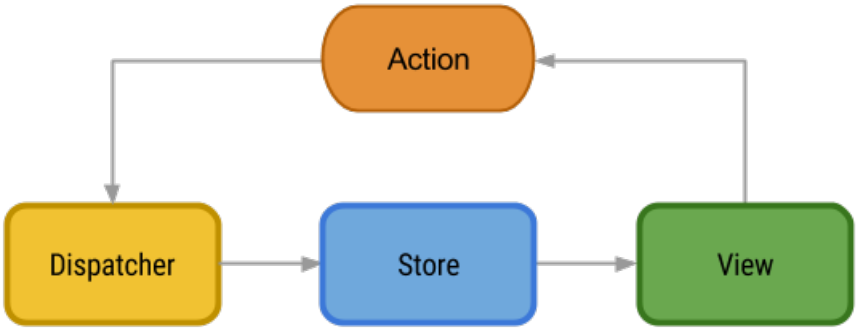


Figura 4.1: Schema sulle componenti di Flux

Flux è un *design pattern* introdotto e reso popolare da Facebook come alternativa al classico modello *MVC* (model-view-controller) giudicato poco adatto ad applicazioni complesse dove sono presenti più di una vista e un modello nella stessa applicazione [65].
Flux è costituito da quattro parti, organizzate come una pipeline di dati unidirezionale strutturata come rappresentato nella figura 4.1:

- **Store:**
  Lo Store è un oggetto che gestisce lo stato, sia quello relativo ai dati che quello relativo all'interfaccia utente, attraverso dei metodi definiti al suo interno.
  Ad esempio nella applicazione implementata sono stati usati due store separati: uno relativo alla parte della connessione fra i due client (*connectionManager*) e uno relativo allo stato di gioco (*gameManager*).

- **Dispatcher:**
  Il Dispatcher è un singolo oggetto che trasmette azioni a tutti gli Store che sono registrati alla ricezione di un determinato evento.

- **Viste:**
  Per Vista (*view*) si intende l'interfaccia utente che si occupa di visualizzare le informazioni contenute negli Store e gestire l'interazione con l'utente.
  Al variare delle informazioni all'interno dello Store, la vista viene aggiornata in modo opportuno.
  Attraverso una Vista è possibile inoltre far partire un'azione in risposta ad una interazione con l'utente.

- **Azioni:**
  Un'Azione (*action*) è un oggetto caratterizzato da un attributo tipo (solitamente una stringa) e un oggetto definito *payload* contenente tutte le informazioni relative all'azione.
  L'unico modo all'interno dell'applicazione per modificare uno stato è attraverso la creazione di un'azione.

Per la nostra applicazione è stato utilizzata la libreria Redux, che adotta le idee di Flux ed è molto spesso accoppiata con React o altri framework simili per interfacce utente.

4.1.3 WebRTC

WebRTC (*Web Real-Time Communication*) è una tecnologia opensource che consente agli utenti di inviare contenuti multimediali (audio e/o video) o dati in tempo reale tramite browser senza la necessità di installare alcun plugin [66]. Grazie a questa tecnologia due utenti possono aprire un canale di comunicazione diretta tra loro, usando una topologia peer-to-peer.
La creazione di questa connessione diretta è preceduta da una procedura detta "*signaling*" che prevede uno scambio di metadati (chiamati "*signal*") tra i due peer come si può vedere nella figura 4.2.

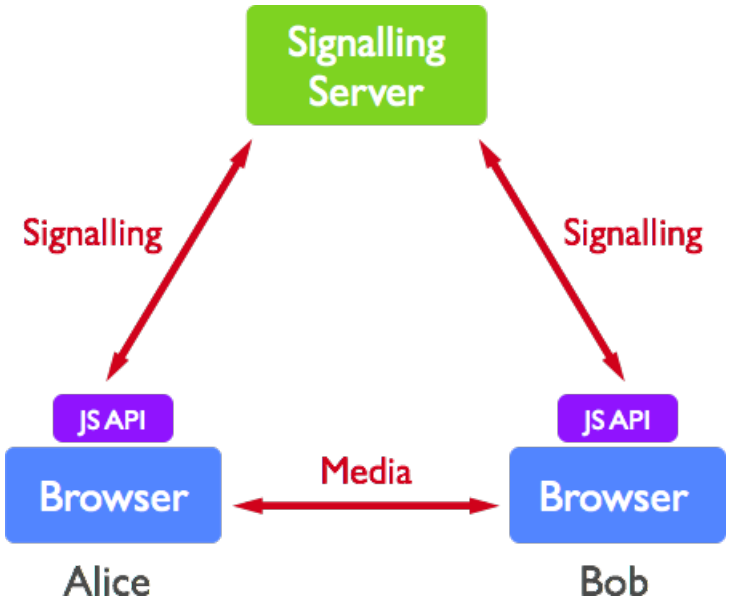


Figura 4.2: Topologia di un semplice collegamento WebRTC

Il protocollo WebRTC non specifica in quale modo i due utenti si debbano scambiare i *signal*, nel nostro caso ci appoggiamo ad un server centrale, *signalhub*. Ci sono due tipologie di connessioni che si possono aprire attraverso le API offerte da WebRTC:

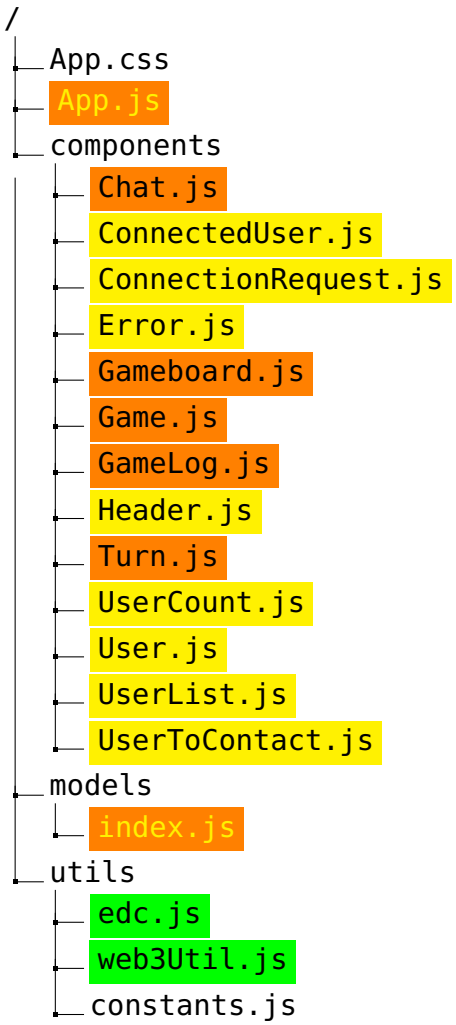
- **MediaStreams**: permettono lo stream di flussi audio e/o video, sono usati ad esempio da applicazioni di videoconferenza.
- **RTCDataChannel**: permettono trasferimenti peer to peer bidirezionali di dati arbitrari.

Nel nostro caso viene creato un RTCDataChannel per scambiarsi messaggi di testo tra i due peer. I dati sono inviati attraverso il protocollo *UDP* in modo sicuro *end-to-end* grazie a *DTLS* [67].

4.1.4 Struttura Client Web

Grazie all'impiego di React è stato possibile strutturate il client web in modo modulare per favorire il riuso del codice. Nel file App.js è definito

il layout generale dell'applicazione e funge da ponte di unione tra la parte dedicata alla connessione e la parte dedicata al Gioco e alla Chat.
Nel file index.js contenuto nella cartella models vengono definiti i due store (connectionManager e gameManager) e le relative azioni che li manipolano.
I file evidenziati in verde (edc.js e web3Util.js) sono sono pensati per essere indipendenti dalla web app finale e dal framework React usato altrove e quindi per essere riusati in altri contesti dove si vuole aprire una comunicazione WebRTC sfruttando Metamask.
I file evidenziati in giallo sono tutti i componenti dell'interfaccia grafica che gestiscono le fasi relativa alla creazione del canale di comunicazione, mentre i file in arancione sono relativi alle funzionalità (Chat e Gioco) disponibili dopo la creazione della connessione.



4.1.5 Protocollo per la creazione del DataChannel

Sfruttando la possibilità di MetaMask di firmare dati con la chiave privata dal proprio wallet, è possibile autenticare l'identità dei due client che si connettono tramite WebRTC, garantendo che i due utenti che vogliono creare un canale di comunicazione tra loro siano gli effettivi proprietari dei wallet Ethereum che dichiarano di controllare.

1. Tramite *Metamask* si ottiene l'indirizzo del wallet del client e si inizia ad ascoltare presso il server *Signalhub* sia in un canale pubblico (identificato come "*publicRoom*") che in uno relativo al proprio indirizzo.

2. Nel canale pubblico ogni 3 secondi viene postato un messaggio in cui si segnala la propria disponibilità associata ad un *timestamp*. Questo permette di popolare una lista di utenti interessati a connettersi tra loro e filtrare gli utenti che non sono più disponibili ignorando i messaggi più vecchi.

3. Quando un utente (detto *iniziatore*) vuole connettersi ad un altro, firma un messaggio con la propria chiave privata dichiarando la propria intenzione, si iscrive al canale privato dell'altro utente e invia il proprio messaggio di richiesta firmato.
   Nel messaggio di invito viene indicato anche un indirizzo ad un wallet temporaneo di servizio, creato all'apertura dell'applicazione, usato per firmare i messaggi successivi senza dover far comparire un prompt di Metamask ad ogni firma.

4. L'altro utente verifica che il messaggio sia stato inviato effettivamente dal mittente corretto e dà una risposta alla richiesta di connessione nel proprio canale.
   In caso di risposta positiva l'utente iniziatore genera il proprio *signal* WebRTC, lo firma e lo invia al secondo utente.

5. L'utente iniziatore, dopo aver verificato la validità del messaggio firmato, invia a sua volta il proprio *signal* WebRTC.

6. Terminato lo scambio dei metadati, il canale WebRTC è aperto, e i due utenti possono accordarsi per l'apertura dello State Channel.

Firma messaggi strutturati tramite lo standard EIP712

L'introduzione della funzionalità di firma di dati attraverso wallet come Metamask ha creato la possibilità di autenticare azioni di un utente senza la creazione di operazioni *on chain*, risparmiando sulle commissioni delle transazioni e migliorando l'user experience di un'applicazione decentralizzata.
Tuttavia inizialmente non era stato pensato nessuno standard per rappresentare i dati strutturati da firmare e quindi Metamask visualizzava nella finestra di conferma della firma il messaggio da firmare come una stringa esadecimale o alternativamente come una stringa di testo.
Questo approccio non rendeva l'uso della firma molto pratico perché gli utenti difficilmente erano in grado di comprendere cosa stessero firmando.
Per risolvere questa problematica nel settembre 2018 è stata approvata la proposta EIP[1]712 [68] che ha introdotto uno standard per definire un messaggi strutturati compatibile con il linguaggio Solidity e la EVM per poter facilmente verificare i messaggi anche on chain.
Occorre definire in un array i nomi dei campi del messaggio associandoli ad un tipo specifico, ad esempio questa è la definizione del tipo del messaggio "Signal" usato nella parte finale del protocollo di connessione per firmare le offerte e le risposte WebRTC.

```
1 const signalType = [{
2       name: 'myAddress',
```

[1] Ethereum Improvement Proposal, proposte di miglioramento relative alla blockchain di Ethereum che vengono discusse pubblicamente dalla comunità per un'eventuale implementazione.

```
3       type: 'address'
4     },
5     {  name: 'myServiceAddress',
6       type: 'address'
7     },
8     {  name: 'targetAddress',
9       type: 'address'
10    },
11    {  name: 'signalData',
12      type: 'string'
13    }]
```

Per impedire che un messaggio firmato possa essere riusato in un altro contesto viene inoltre definito un dominio che serve a limitarne la validità solamente all'interno di una determinata dApp. All'interno del dominio è possibile specificare il nome dell'applicazione, la blockchain dove è operativa ed altri parametri univoci (come l'indirizzo dello SmartContract).

```
1  const domainType = [{
2       name: "name",
3       type: "string"
4    },
5    {
6       name: "version",
7       type: "string"
8    },
9    {
10      name: "chainId",
11      type: "uint256"
12   },
13   { name: "verifyingContract",
14     type: "address"
15    }
16 ];
17 const domainData = {
18     name: "Ethereum WebRTC Game",
```

```
19      version: "2",
20      chainId: 3,
21      verifyingContract: "0x1C56346CD2A2Bf3202F771f50d3D14a367B48070"
22  };
```

Nella figura 4.3 è illustrato un esempio di prompt di Metamask per la firma di un messaggio dove vengono mostrati all'utente in modo chiaro sia i dati che sta firmando che il dominio riferito a quei dati.

Figura 4.3: Esempio di firma di messaggio.

4.2 Implementazione State Channel

La parte del progetto relativa agli smart contract si può dividere in tre parti che interagiscono fra loro.

1. Lo smart contract dello State Channel, che gestisce l'apertura e la chiusura dei canali e l'apertura delle dispute.
   Inoltre all'interno del contratto è implementata un semplice esempio di banca (FreezeBank) per consentire agli utenti di bloccare dei fondi all'apertura dello State Channel.

2. Lo smart contract della State Machine, responsabile della manipolazione dello stato relativo all'applicazione.
   È composto da due contratti separati, uno dei quali definisce solamente l'interfaccia per consentire potenzialmente l'uso di implementazioni di State Machine differenti.

3. Lo smart contract riferito al gioco scelto come esempio. Anche in questo caso possiamo vedere due contratti separati: un'interfaccia che descrive le funzioni che un contratto di gioco deve avere per essere compatibile con lo State Channel e un contratto che tramite questa interfaccia implementa le regole di gioco del gioco *Pig*, scelto come esempio per il nostro progetto.

Inoltre è stato implementato sotto forma di *libreria*, uno smart contract di funzioni e tipi comuni, *Common*, usato per definire le strutture dati, riferite agli stati e alle firme, in comune fra i contratti che le manipolano.

4.2.1 Strumenti utilizzati

- **Truffle**:
  Framework di sviluppo per blockchain basate sulla EVM, che permette di compilare, testare e fare deploy degli smart contract sviluppati in modo semplice.

- **GanacheCLI**:
  Simulatore di blockchain Ethereum che permette di testare gli smart contract durante lo sviluppo in locale senza dover installare o connettersi ad un nodo Ethereum completo.

Il framework Truffle gestisce in automatico il download della versione desiderata del compilatore *Solidity*, *solc*, nel nostro caso abbiamo usato la versione 0.5.10 con l'ottimizzatore attivato per minimizzare le dimensioni del bytecode generato.

4.2.2 Gerarchia degli Smart Contract

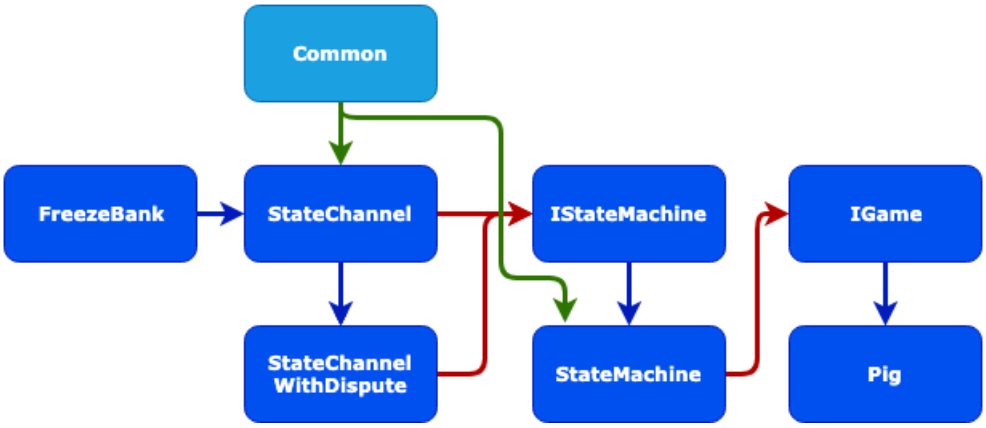


Figura 4.4: Grafico delle relazioni tra i vari smart contract.

Nella figura 4.4 ogni riquadro colorato rappresenta un file contenente codice Solidity, l'azzurro indica le librerie mentre il blu indica gli Smart Contract veri e propri.
Le frecce colorate rappresentano le diverse relazioni tra i vari contratti: il verde indica l'inclusione del file come libreria, il blu indica che i due contratti sono in relazione tra loro per ereditarietà, mentre il rosso indica che il primo contratto fa delle chiamate esterne verso il secondo.
I contratti *IStateMachine* e *IGame* sono utilizzati come interfacce: al loro interno sono definiti solamente i prototipi delle funzioni necessarie

per implementare una State Machine ed un Gioco, mentre il contenuto delle funzioni è implementato nei contratti che ereditano le interfacce. In questo modo i contratti che fanno chiamate esterne verso queste due tipologie di contratto,non dipendono da una singola implementazione specifica del contratto.
Per ragioni di sicurezza, con la EIP170 [69] il bytecode generato di un singolo contratto non può superare la dimensione di 24kb.
Con l'implementazione di tutte le funzioni relative alle dispute, il contratto relativo agli State Channel superava questo limite impedendo il processo di deploy in una blockchain di test.
Per questa ragione è stato necessario implementare la classe Common come libreria anziché come contratto per ridistribuire il bytecode generato su contratti diversi.

4.2.3 ABIV2Encoder

Tutti i contratti implementano una funzione sperimentale di Solidity, ABIV2Encoder attivata all'inizio del contratto con la direttiva `pragma experimental ABIEncoderV2` [70].
Questa funzione consente di strutture, variabili nidificate e dinamiche come parametri o come valori da ritornare all'interno della funzione, consentendo di passare facilmente da un contratto all'altro le strutture rappresentanti gli stati dei diversi contratti.

4.2.4 State Machine e State Channel

Ricordiamo che possiamo definire la State Machine come un contratto che implementa che implementa la funzione che descriva una transizione di stato:

```
1 advanceState(state, action) => newState
```

Questa funzione ha come parametri uno stato ed un'azione che agisce su di esso e al termine della computazione restituisce un nuovo stato. È importante che questo metodo non abbia effetti collaterali, dovendo essere deterministico come qualsiasi procedura eseguibile su blockchain. Possiamo quindi definire uno State Channel come un protocollo che permette a due o più partecipanti di aggiornare uno stato condiviso attraverso una *State Machine,* assicurando che ciò possa avvenire in modo cooperativo *offchain,* ma con le stesse proprietà di sicurezza che avremmo se l'interazione avvenisse direttamente sulla blockchain.
Prima dell'apertura del canale è necessario che alcuni fondi siano impegnati da entrambi i partecipanti.
Successivamente lo stato verrà aggiornato interattivamente *off chain,* fino a quando i due utenti concorderanno su uno stato finale, in base al quale interagendo con uno smart contract *on chain* verranno restituiti i fondi precedentemente congelati.

4.2.5 Implementazione FreezeBank

Il contratto FreezeBank è utilizzato per bloccare all'apertura dello state channel i fondi dei due partecipanti e al termine dell'interazione tra i partecipanti ritirare i fondi sbloccati.
Lo scheletro del contratto è il seguente:

```
1 mapping (address => uint256) internal balances;
2 mapping (address => uint256) internal frozenBalances;
3
4 function getBalance() public view returns (uint)
5 function deposit() public payable returns (uint)
6 function withdraw(uint256 withdrawAmount) public returns (uint)
7 function freeze(address target, uint frozenAmount) internal returns
    (uint)
8 function unFreezeSplit(address p1, address p2, uint256 b1, uint256 b2)
    internal returns(bool)
```

Possiamo notare due strutture dati di tipo *mapping*, una che tiene traccia dei bilanci depositati da ciascun indirizzo e un'altra per i bilanci congelati in seguito all'apertura di un canale.
Le prime tre funzioni sono le classiche funzioni di una banca che permettono di depositare, ritirare fondi e conoscere il proprio bilancio e sono utilizzabili da chiunque, avendo una visibilità pubblica, mentre le ultime due sono invocabili solamente dal contratto stesso essendo dichiarate come *internal*:

- **Freeze** è richiamata all'apertura dello State Channel per ciascun partecipante, e si occupa di congelare i fondi di un utente sottraendoli dal *mapping balances*, impedendogli in questo modo di ritirarli. Ovviamente i fondi che si vogliono bloccare devono essere stati precedentemente depositati.

- **unFreezeSplit** è richiamata alla chiusura del canale e ripartisce il totale congelato dalla funzione *freeze* in base al valore indicato dai due parametri *b1* e *b2*.

Per motivi di sicurezza e di efficienza nella funzione *unFreezeSplit* i fondi non sono direttamente inviati ai due indirizzi rispettando il design pattern *withdrawal* [71].
Il *withdrawal pattern* cede la responsabilità per la riscossione di fondi, sul destinatario che deve inviare una transazione per ottenerli.
L'uso di questo *design pattern* semplifica molto lo sviluppo del contratto che non deve gestire più gestire i casi dove l'invio dei fondi fallisce.

4.2.6 Apertura dello State Channel

Dopo che i due utenti hanno depositato la propria quota, ad esempio 1 *ether*, possono iniziare ad accordarsi per aprire lo State Channel.
La funzione openChannel() ha questo prototipo:

```
1 function openStateChannel(bytes memory packedOpenChannelData) public
    returns (bool)
```

Il parametro packedOpenChannelData ha come tipo StateChannelOpenData e viene *compresso* prima dell'invio come parametro della funzione al nodo Ethereum.

```
1    struct StateChannelParticipant {
2       address participantAddress;
3       address signingAddress;
4       uint256 amount;
5    }
6    struct StateChannelOpenData {
7       bytes32 channelID;
8       address channelAddress;
9       StateChannelParticipant[2] participants;
10      address stateMachineAddress;
11      uint256 timeStamp;
12      bytes32 initialStateHash;
13      Signature[2] initialStateSignatures;
14      bytes packedStateMachineInitialisationData;
15   }
```

- *ChannelID* rappresenta un identificativo univoco associato allo State Channel che ci si appresta a creare.
- *ChannelAddress* indica l'indirizzo del contratto relativo allo State Channel stesso, viene incluso per sicurezza per evitare *replay attack* (vedi sezione 4.2.9).
- Nell'array *participants* sono specificati i dati relativi ai due partecipanti del canale: per ogni partecipante associamo due diversi indirizzi.
  Il primo è l'account principale di Ethereum, usato per firmare la richiesta iniziale di connessione e la richiesta successiva di apertura del canale, il secondo è invece un indirizzo di servizio, la cui chiave privata viene generata a *runtime*, che viene utilizzato per

firmare gli stati successivi senza la necessità di chiedere continue conferme all'utente.
Nella funzione *validateSignature* che verifica se uno stato sia firmato correttamente vengono usati sequenzialmente entrambi gli indirizzi per permettere ad utente di firmare uno stato anche nel caso perda l'accesso alla chiave privata dell'account di servizio generato all'apertura dell'applicazione.
La variabile *amount* associata ad ogni partecipante indica quanti *ether* sono stati impegnati nello stato inizialmente.

- La variabile *timeStamp* è usata per limitare il tempo per cui lo StateChannelOpenData è valido.
  Di default è possibile aprire lo stato fino ad un'ora dopo che i due partecipanti si sono accordati e hanno firmato lo stato.

- *StateMachineAddress* indica l'indirizzo del contratto *State Machine* che sarà usato dai due partecipanti per aggiornare lo stato.

- In *initialStatehash* viene salvato l'hash dello stato iniziale fornito dalla State Machine e l'array *initialStateSignatures* contiene le firme relative a questo stato dei due partecipanti.

- *PackedStateMachineInitialisationData* contiene i dati compressi di tipo *StateMachineInitialisationData* (descritto nella prossima sezione), che servono ad ottenere lo stato iniziale.

Struttura dello Stato

Lo stato dello State Channel è stutturato in questo modo:

```
1  struct State {
2     StateContents contents;
3     Signature[2] signatures;
4  }
5  struct StateContents {
6     bytes32 channelID;
```

```
7       address channelAddress;
8       uint256 nonce;
9       bytes packedStateMachineState;
10   }
```

Viene nuovamente indicato il *channelID* e l'indirizzo del contratto del canale, è presente un nonce e lo stato della State Machine codificato tramite *ABIv2Encoder*.
Memorizzando lo stato in questo modo, pur non essendo a conoscenza della struttura dati della State Machine, il contratto dello State Channel può comunque verificare se lo stato è firmato correttamente per poi passarlo alla State Machine quanto è necessario eseguire una transizione di stato.

Stato della State Machine

La stato della State Machine ha questa struttura:

```
1    struct StateMachineState {
2       address gameContract;
3       address[2] SigningAddress;
4       bytes32[2] previousSeeds;
5       bool isOver;
6       bytes packedGameState;
7       uint[2] scores;
8       uint8 nextPlayer;
9       bool hasPendingPAction;
10      bytes packedPAction;
11   }
12   struct StateMachinePAction {
13      uint256 pNumber;
14      bytes32 seed;
15      bytes packedGameAction;
16      Common.Signature packedGameActionCoSignature;
17   }
```

- *GameContract* indica l'indirizzo del contratto che descrive le regole del gioco.
- *SigningAddress,* array dove vengono indicati gli indirizzi dei due giocatori.
- Nell'array *previousSeeds* vengono indicati gli hash usati come seed per generare il numero casuale precedente, secondo il meccanismo descritto nella sezione 3.3.1.
- Il booleano *isOver* indica se la partita è terminata, serve ad impedire di effettuare nuove azioni.
- *PackedGameState* contiene lo stato di gioco compresso secondo la struttura definita nel contratto del singolo gioco.
- L'array *scores* contiene i punteggi aggiornati dei due giocatori.
- *NextPlayer* indica quale giocatore può effettuare la prossima mossa, essendo aggiornato dal contratto di gioco, permette di implementare giochi i giocatori fanno più di un turno di fila, come nel nostro caso.
- Il booleano *hasPendingPAction* indica se la mossa proposta è stata completata: è uguale a *true* se l'avversario deve rispondere all'azione proposta fornendo il proprio seed casuale.
- *packedPAction* contiene le informazioni sull'azione proposta dal giocatore secondo la struttura *StateMachinePAction*: al suo interno indica il numero del giocatore, il seed casuale, la struttura dell'azione di gioco proposta (definita all'interno del contratto di gioco) e la firma di quest'ultima.

Per generare lo stato iniziale, necessario per aprire il canale, viene richiamata la funzione presente nel contratto State Channel:

```
function getPackedInitialStateContents(address stateMachineAddress,
    bytes32 channelID, bytes memory
    packedStateMachineInitialisationData) public view returns (bytes
    memory packedInitialStateContents)
```

Questa funzione ha come parametri l'id del canale scelto ed un oggetto di tipo StateMachineInitialisationData che indica l'indirizzo del contratto di gioco scelto, gli indirizzi dei due giocatori e i seed di partenza ovvero gli ultimi hash generati nella lista di hash usata per generare i numeri casuali.

```
1   struct StateMachineInitialisationData {
2      address gameContract;
3      address[2] SigningAddress;
4      bytes32[2] seeds;
5   }
```

Il nonce dello stato iniziale viene impostato a zero, e viene richiamata la funzione *getInitialPackedStateMachineState* della State Machine per ottenere lo stato iniziale codificato dall'ABI Encoder.
La funzione *getPackedInitialStateContents* è una view pubblica: in questo modo i client possono quindi richiamarla gratuitamente *offchain* per generare lo stato iniziale da firmare.

State Channel nello Storage

Dopo aver verificato che lo stato iniziale indicato sia stato correttamente firmato da entrambi i giocatori, all'interno della funzione *openStateChannel* si procede a memorizzare nello *Storage* del contratto il riferimento al canale appena aperto.

```
1   enum SCState {Void,Open,Closed,Disputed}
2   struct StateChannelData {
3      SCState channelStatus;
4      bytes32 packedOpenChannelDataHash;
```

```
5     }
6     mapping (bytes32 => Common.StateChannelData) public
          stateChannelData;
```

All'identificativo di ciascun canale corrisponde un oggetto di tipo StateChannelData che contiene lo status del canale (*channelStatus*), impostato inizialmente uguale ad *Open*, che ha come tipo l'enumerazione *SCState* e l'hash *packedOpenChannelDataHash* dell'oggetto StateChannelOpenData usato per aprire il canale.
Salvare solamente l'hash anziché l'intera struttura permette di rendere la transazione di creazione del canale molto più economica e i dati relativi al canale sono in ogni momento ricavabili a partire dalle informazioni contenute in uno *stato*.
Al termine dell'azione di apertura dello State Channel viene emesso l'evento *StateChannelOpened*.

4.2.7 Aggiornamento dello Stato

Arrivati a questo punto abbiamo uno State Channel aperto e ogni giocatore possiede una copia dello stato iniziale firmata da entrambi.
Per poter cambiare lo stato, idealmente senza effetturare transazioni su blockchain dobbiamo introdurre il concetto di *azione*.

```
1     enum ActionType {Advance,Close}
2     struct ActionContents {
3        bytes32 channelID;
4        address channelAddress;
5        uint256 stateNonce;
6        uint8 participant;
7        ActionType actionType;
8        bytes packedActionData;}
9     struct Action {
10       ActionContents contents;
11       Signature signature;}
```

In questa struttura è indicato l'identificativo del canale e l'indirizzo del contratto (per le solite ragioni di sicurezza), il nonce dello stato a cui va applicata la transazione e il partecipante che ha proposto l'azione.
Si distinguono due tipologie diverse di azione, *Advance* e *Close*, per indicare rispettivamente l'intenzione di aggiornare lo stato del canale oppure proporre di chiuderlo, terminando la partita.
Il contratto dello State Channel espone una funzione pubblica, di sola lettura, che prende uno Stato, un'azione e l'indirizzo di una State Machine, e che restituisce un nuovo Stato: questa funzione è chiamato *offchain* dai partecipanti per far avanzare lo stato in modo deterministico.

```
1 function advanceState(StateContents memory stateContents,
    ActionContents memory actionContents, IStateMachine stateMachine)
    public view returns (bool isValid, StateContents memory
    newStateContents)
```

Questa funzione al suo interno fa una chiamata alla funzione omonima contenuta nella State Machine per aggiornare lo stato, ritornandone uno nuovo con il nonce incrementato.
La State Machine verifica che l'azione proposta sia valida per lo stato fornito, verificando ad esempio che l'azione indicata non infranga le regole del gioco e rispetti l'ordine di turno.
Per ogni azione proposta da un giocatore viene richiamata la funzione *advanceState* due volte:

1. Il giocatore propone l'azione includendo il proprio *seed* casuale: il nuovo stato generato è uguale al stato precedente, ma contenente al suo interno l'azione proposta e il booleano *hasPendingPAction* è impostato a *true*.

2. L'avversario risponde includendo nell'azione di risposta il proprio *seed*: viene effettivamente calcolato lo stato successivo, chiamando la funzione *playMove* del contratto di gioco passando come

parametri lo stato di gioco precedente, il *seed* casuale calcolato come hash della concatenazione dei seed dei due giocatori e l'azione proposta.

In entrambi i passaggi viene inoltre verificata la validità del seed proposto dal giocatore, calcolandone l'hash e confrontandolo con quello inviato nel turno precedente.

4.2.8 Chiusura del canale

Affinché un canale di Stato possa essere chiuso, entrambi i partecipanti devono essere concordare firmando un'azione di tipo *Close* indicando il nonce dello stato finale su cui hanno concordato.
Le due azioni, insieme allo stato finale firmato da entrambi sono passate alla funzione di chiusura:

```
1 function closeStateChannel(bytes memory packedOpenChannelData, State
    memory state, Action[2] memory actions) public
```

Per poter chiudere con successo un canale il contratto effettua una lunga serie di controlli:

- L'hash del *packedOpenChannelData* indicato come parametro deve essere uguale a quello salvato nello storage del canale.
- Lo status dello State Channel deve essere "Open" oppure "Disputed". Se entrambi i partecipanti concordano è possibile chiudere uno stato in disputa.
- Lo stato indicato deve essere valido: viene verificata la validità delle variabili *channelAddress* e *channelID*, e che lo stato sia firmato da entrambi i partecipanti.
- Entrambe le azioni sono valide: devono essere entrambe firmate, avere il tipo Close ed le variabili *channelAddress*, *channelId* e *stateNonce* devono essere corrette.

Se tutti questi controlli hanno successo allora il canale può essere chiuso, e la funzione per ridistribuire i fondi bloccati all'apertura viene richiamata.

```
1 function DistributeFundsAndCloseChannel(bytes memory
      packedOpenChannelData, Common.State memory state, uint256 penalty)
```

Questa funzione imposta lo status dello State Channel memorizzato nello Storage del contratto a "*Closed*" e richiama la funzione *getPayouts* della State Machine che, in base alla regole stabilite nel contratto del gioco, ritorna i nuovi bilanci dei due partecipanti.
Questa funzione prevede la possibilità che possano essere applicate delle penalità alla ricompensazione dei due giocatori, ma se il canale viene chiuso in comune accordo nessuna penalità viene applicata.
Infine, dopo aver scongelato i fondi dei due giocatori attraverso la funzione *unFreezeSplit*, viene emesso l'evento StateChannelClosed.

4.2.9 Prevenire i replay attack

Nelle varie strutture analizzate abbiamo incluso in maniera apparentemente ridondante informazioni relative all'indirizzo del contratto usato, agli indirizzi dei partecipanti, al nonce riferito allo stato e ad altri dati simili.
Questa ripetizione di dati in realtà è fondamentale per evitare eventuali *replay attack*: un avversario infatti potrebbe conservare un messaggio firmato da un utente in un altro contesto e riusarlo in seguito per imbrogliare, ad esempio, in una disputa.
La protezione a queste tipologie di attacco è già prevista nelle transazioni Ethereum classiche ma è stato necessario pensare tutta una serie di controlli da effettuare ad ogni passaggio per garantirla in maniera analoga negli aggiornamenti di stato effettuati *offchain*.

4.2.10 Gestione delle dispute

Quando un partecipante non è in grado di aggiornare lo stato del canale perché l'altro partecipante non segue più il protocollo descritto nella sezione 3.3, può far partire il processo di *disputa*.
Per ragioni pratiche, tutto il codice che gestisce gli svariati casi di disputa è contenuto in un file separato nel contratto *StateChannelWithDispute* che eredita tutte le funzioni del contratto descritto fin'ora.
La risoluzione di una disputa consiste nell'invocare la blockchain per costringere il giocatore avversario a comportarsi di nuovo correttamente: il risolversi di una disputa può portare alla chiusura dello State Channel oppure al proseguimento della partita.
Le scheletro della parte di contratto dedicata alle dispute è il seguente:

```
1  struct DisputeData {
2     uint256 startTime;
3     uint256 initiator;
4     uint256 stateNonce;
5     bytes32 stateContentsHash;
6     bytes32 actionContentsHash;
7  }
8  mapping (bytes32 => DisputeData) public disputeData;
9  enum DisputeType {NoAction, Action}
10 enum ResolveType {WithAction, AgreeAndCloseChannel, TimeOut,
       LaterState, DifferentAction}
11 function openDispute(DisputeType dtype, bytes memory
       packedOpenChannelData, State memory state, Action memory action,
       State memory proposedNewState) public returns(bool)
12 function resolveDispute(ResolveType rtype, bytes memory
       packedOpenChannelData, State memory state, Action memory action,
       State memory proposedNewState) public returns(bool)
```

La situazione più tipica per cui può iniziare una disputa è il caso in cui un partecipante è in possesso di uno stato firmato da entrambi i partecipanti e vuole generare un nuovo stato richiamando la funzione

advanceState offchain.
Tuttavia per una qualsiasi ragione la controparte non risponde controfirmando lo stato proposto e il partecipante rimane bloccato.
Per ottenere uno stato aggiornato è quindi possibile aprire una disputa chiamando la funzione *openDispute* specificando nel parametro *dtype* la tipologia di disputa *Action*.
Ogni volta che si apre una disputa, lo smart contract esegue una lunga serie di controlli:

- Lo *State Channel* deve essere aperto senza nessuna ulteriore disputa in corso.
- Viene controllato che lo stato e l'azione indicati siano validi nel contesto del canale in maniera analoga al caso della chiusura del canale descritto nella sezione 4.2.8.
- Lo stato precedente fornito deve essere correttamente firmato da entrambi i partecipanti.
- L'azione proposta deve essere correttamente firmata dal partecipante che inizia la disputa e deve essere di tipo "*Advance*".
- Il *nonce* dello stato indicato deve essere maggiore di quello di un'eventuale disputa precedente.

Infine si genera il nuovo stato *on chain* richiamando la funzione *advanceState* e si verifica che lo stato generato corrisponda a quello proposto dal partecipante che vuole aprire la disputa.
Se tutti i controlli passano, possiamo cambiare lo status del canale in "*Disputed*" e memorizziamo nello Storage su blockchain alcuni dati relativi alla disputa: chi l'ha iniziata, l'hash del nuovo stato e dell'azione proposta, il nonce del nuovo stato, e un timestamp.
Attraverso la funzione *resolveDispute* viene chiusa una disputa precedentemente aperta.

Possiamo distinguere 5 casistiche di risoluzione diverse a seconda del parametro *rtype* di tipo *ResolveType* specificato:

- **WithAction:**
  La controparte della disputa concorda sullo stato proposto e risponde fornendo la propria prossima azione firmata, la firma sullo stato proposto precedentemente e un nuovo stato risultante dall'azione proposta firmato.
  Vengono fatti controlli simili a quelli fatti precedentemente per aprire la disputa: l'azione proposta deve essere valida e del tipo corretto, il nuovo stato proposto deve essere correttamente firmato e valido.
  Questa risoluzione ha come risultato il proseguimento della partita in corso.

- **AgreeAndCloseChannel:**
  La controparte oltre a concordare sullo stato proposto come nel caso *WithAction* decide di chiudere il canale.
  Questa casistica è fondamentale nel caso in cui l'azione proposta nella disputa porterebbe ad uno stato finale che non prevede altre azioni valide.
  In questo caso se l'azione proposta dalla controparte è di tipo *End* e tutti i controlli descritti per il caso precedente hanno successo, lo state channel è chiuso senza alcuna penalità.

- **TimeOut:**
  Se la controparte non risponde all'apertura della disputa in un intervallo di tempo ragionevole, ad esempio 2 ore, lo State Channel può venir chiuso, su richiesta dell'iniziatore, penalizzando la controparte che non ha risposto. L'entità della penalizzazione è decisa nel contratto di gioco.

- **LaterState:**
  Se l'iniziatore prova ad aprire una disputa fornendo uno stato pre-

cedente all'ultimo su cui si sono accordati i due partecipanti, la controparte può rispondere fornendo uno stato firmato da entrambi i partecipanti con un *nonce* maggiore e vincere la disputa.
Aprire una disputa con uno stato invalido è molto sconveniente poiché la controparte può rispondere con un qualsiasi stato successivo e quindi idealmente sceglierà quello più vantaggioso nei suoi confronti.
Lo state channel viene quindi chiuso, penalizzando l'iniziatore della disputa.

- **DifferentAction:**
  Un partecipante potrebbe firmare una transizione di stato proponendo una certa azione ed inviarla alla controparte, per poi successivamente aprire una disputa usando lo stesso stato ma abbinandogli un'azione differente.
  Ciò potrebbe rivelarsi vantaggioso per come funziona la generazione di numeri casuali nella nostra implementazione di *state channel*, il partecipante potrebbe attendere la risposta della controparte e decidere di cambiare l'azione in base all'esito dell'azione.
  In questa caso, la controparte può rispondere fornendo la prima azione firmata accompagnata dallo stato. Se i controlli di validità opportuni hanno successo, lo state channel viene chiuso e l'iniziatore della disputa viene penalizzato.

Un'altra tipologia di disputa possibile è un caso particolare della situazione descritta precedentemente.
Questa disputa, del tipo *NoAction* avviene se l'avversario risponde ad un'azione fornendo la firma sul nuovo stato proposto ma, nonostante tocchi a lui proporre una nuova azione, non proponendo un nuovo stato successiva. In questo caso si usa la firma fornita per aprire una disputa e attendere un'eventuale risoluzione analoga a quelle già descritta da parte della controparte.

4.2.11 Implementazione regole di gioco

Nell'interfaccia IGame sono descritte le funzioni che devono essere implementate in un qualsiasi gioco:

```
1 contract IGame {
2    function getPackedInitialGameState() public view returns (bytes
         memory packedInitialGameState);
3    function validateMove(bytes memory packedGameState, bytes memory
         packedGameAction) public view returns (bool isValid);
4    function playMove(bytes32 rngSeed, bytes memory packedGameState,
         bytes memory packedGameAction) public view returns ( uint256
         nextPlayer, bytes memory packedNewGameState, bool isOver,
         uint256[2] memory score);
5    function calculatePayouts(uint256 balance0, uint256 balance1,
         uint256 score0, uint256 score1, uint8 penalty) public view
         returns (uint256 newBalance0, uint256 newBalance1);
6 }
```

- *GetPackedInitialGameState()*: Ritorna lo stato di gioco iniziale compresso. Questa funzione è richiamata dalla State Machine quando viene generato lo stato iniziale dello State Channel.

- *ValidateMove()*: Dato uno stato di gioco e un'azione di gioco, restituisce un booleano che indica la validità dell'azione proposta.

- *PlayMove()*: Dato uno stato di gioco, un'azione e un seed casuale, restituisce un nuovo stato, il punteggio aggiornato, un booleano che indica se si è raggiunto uno stato finale e un intero che indica il prossimo giocatore che dovrà muovere.

- *CalculatePayouts()*: Fornendo i bilanci iniziali bloccati dai due giocatori, il punteggio finale raggiunto alla fine della partita e un intero che indica l'eventuale giocatore penalizzato, calcola i premi dei due giocatori.

Implementazione gioco Pig

Lo stato del gioco implementato *Pig*, ha la seguente struttura:

```
1 struct State {
2    uint256[2] score;
3    uint256 partialScore;
4    bool firstAction;
5 }
6 struct ActionType {
7    bool play;
8    uint256 pNumber;
9 }
```

Lo stato indica il punteggio totale dei due giocatori, il punteggio parziale del round del giocatore di turno e un booleano che indica se l'azione da svolgere è la prima.
L'azioni possibili sono due, definite dai due stati del booleano *play*: è possibile decidere di tirare il dato oppure fermarsi se si è tirato il dado almeno una volta.
La funzione *calculatePayouts* concede tutto il montepremi ad uno dei due giocatori se questo ha raggiunto i 100 punti oppure se l'avversario è stato penalizzato in seguito ad una disputa.
Altrimenti, se il gioco è stato interrotto senza nessuna penalità, i premi sono divisi proporzionalmente ai due punteggi.

Capitolo 5

Validazione ed Esperimenti

In questo capitolo vengono illustrati i risultati di alcuni misurazioni eseguiti con gli smart contract sviluppati.
Dopo aver testato il funzionamento degli smart contract in locale, sono state effettuate ulteriori prove in una rete di test pubblica (*test net*) per misurare le prestazioni in un ambiente più simile a quello reale.
Per ogni possibile metodo si è misurato il gas speso e il tempo necessario per approvare ciascuna transazione. Infine partendo dal costo in gas calcolato, basandoci su delle statistiche pubbliche abbiamo stimato i tempi necessari e i costi che si dovrebbero affrontare sulla rete principale Ethereum.
Infine, a scopo indicativo, i prezzi delle transazioni in *Ether* sono stati convertiti in Euro, considerando come tasso di conversione $1 ETH \cong 155$€, aggiornato ad Ottobre 2019 [72].

5.1 Deploy

Ci sono svariate reti di test pubbliche su cui poter testare i propri smart contract prima di effettuare il deploy nella rete Ethereum principale, ciascuna con diverse caratteristiche [73].
Per i nostri esperimenti abbiamo scelto la rete di test *Ropsten*.

Questa scelta è stata motivata dal fatto che è quella più simile alla rete principale: mentre le altre *test net* sono centralizzate per motivi di stabilità, *Ropsten* analogamente alla *main net* è basata sempre su un algoritmo di consenso *proof of work*.
I tempi di validazione tra un blocco e l'altro sono quindi simili a quelli reali, anche se di contro è più vulnerabile ad attacchi di tipo *denial of service* [74].
Per ottenere *Ether* da impiegare nella test net è possibile estrarli, installando un nodo Ethereum e iniziando a validare blocchi di transazioni, oppure richiederli a servizi web che li regalano [1].
Per interfacciarci alla rete di test senza installare un nodo abbiamo sfruttato il servizio offerto da Infura[62], che, previa registrazione, fornisce un endpoint che può essere indicato nel file di configurazione di Truffle per consentire il deploy.

5.1.1 Transazioni ed Indirizzi

Di seguito sono elencate, in ordine, gli hash delle transazioni di *deploy* e gli indirizzi generati per ciascun contratto.
Tutte le transazioni ed indirizzi citati sono ricercabili attraverso il servizio relativo a Ropsten offerto dal sito web Etherscan[2].

Common:
0x55da02008d0bfaa25c5695a0d3c00eb9714ab8b096dc8d44df9afe41608bf256
0x381fD2AaE5135B036a02d82dA9B9322BFcAFaa37
StateChannelWithDispute:
0x3bd0ae569cb432d49efb58103512cb0b69a746ab47b34f64cc5c79440d99a1f9
0xE1792A86CD8d19a083B00178E4c62eF38C60a314
StateMachine:
0x4f820fa926fb08adb8f81bd2f3239696f5bc1f480bf807cc08633793c17aedab

[1] Questi servizi web sono chiamati in gergo *faucet*, nel nostro caso abbiamo richiesto *ether* presso `https://faucet.metamask.io/`.

[2] `https://ropsten.etherscan.io/`

0xc2d3d03579f42754d7578BF787950640CfdE4a4B

Pig:

0x71d54a06f98f2f6377c3e993878923b40ba955914fe3a2358e5eb2cd20db7165

0x5848C1fD37aDC2eA594225B8CFE96494E4C420E1

5.1.2 Costi e tempi di deploy

Nella tabella 5.1 possiamo osservare dettagli relativi alle tempistiche e ai costi del deploy dei contratti nella test net Ropsten.
Il prezzo per unità di gas è stato fissato per tutte le transazione a 20 gwei (0.00000002 ether) per maggiore velocità di deploy.

Contratto	Tempo (s)	Gas Usato	Costo ETH	Costo €
Common	21	797446	0.01594892	2.47
StateChannelWithDispute	153	4935964	0.09871928	15.30
StateMachine	169	2056654	0.04113308	6.38
Pig	65	828865	0.0165773	2.57
Totale	408	8618929	0.17237858	26.72

Tabella 5.1: Costi e tempi per il deploy dei contratti implementati.

5.2 Prove sulle funzioni

Per le misurazioni è stato utilizzato NodeJS per implementare uno script da linea di comando che ci ha permesso testare le funzioni usate per interagire con gli state channel in modo sistematico.
Il prezzo del gas è stato fissato sempre a 20 *gwei,* dopo aver appurato nei primi test che con quantità offerte inferiori i tempi di approvazione di ciascuna transazione superavano di molto il minuto.

Funzione	Gas	Costo ETH	Costo €
deposit()	29005	0.0005801	0.09
withdraw()	38152	0.00076304	0.12
openChannel()	160923	0.00321846	0.50
closeChannel()	176433	0.0035287	0.55
openDispute()	230362	0.00460724	0.71
resolveDispute()	176980	0.0035396	0.55

Tabella 5.2: Costi dei principali metodi del contratto StateChannelWithDispute.

Nella tabella 5.2 non sono state indicate le tempistiche di approvazione di ciascuna transazione perché non sono risultati influenzati dal differente costo di gas. I tempi misurati ad ogni prova sono risultati molto variabili, tra i 5 e i 40 secondi con una media intorno ai 15 secondi. Il consumo del gas è risultato abbastanza consistente tra un esecuzione e l'altra, con alcune piccole variazioni che sono state considerate facendo una media tra le varie esecuzioni.
Consultando *EthGasStation*[3], un servizio web usato per calcolare i prezzi e i tempi delle transazioni relative alla rete principale di Ethereum, è risultato che tutte le transazioni sarebbero approvabili entro 2 blocchi (circa 27 secondi).
EthGasStation stima il tempo delle transazioni in base al costo in gas e al prezzo del gas tramite un modello di regressione di Poisson basato sui dati degli ultimi 10000 blocchi [75].
I costi di ciascuna funzione risultano abbastanza contenuti, anche se sarebbe opportuno cercare di ridurre il costo delle funzioni relative alle dispute: idealmente aprire una disputa non dovrebbe costare più che chiudere un canale, mentre in questo caso il costo è quasi doppio.

[3] `https://ethgasstation.info/`

5.3 Confronto con soluzione onchain

Per valutare la convenienza nell'uso degli state channel abbiamo stimato il costo di una partita di *Pig* se questa venisse svolta interamente su Blockchain e lo abbiamo confrontato con quello della nostra soluzione. Per fare ciò è stato implementato un altro contratto PigOnChain, che memorizza lo stato di gioco di una partita nello storage della blockchain e permette di modificarlo attraverso transazioni *on chain*.

5.3.1 Descrizione PigOnChain

Usando come base il contratto di Pig, abbiamo aggiunto delle funzioni che gestiscono l'inizio e la fine di una partita e un sistema di generazione di numeri casuali attraverso il meccanismo di commit-reveal analogo a quello usato nella nostra soluzione basata su state channel.
Lo scheletro del nuovo contratto risultante è il seguente:

```
1   mapping(bytes32 => GameInfo) private gameRecords;
2   mapping (bytes32 => State) private gameStates;
3   struct GameInfo {
4      bytes32 gameId;
5      address[2] participants;
6      uint256 joinedPlayers;
7      int8 gameStatus;
8   }
9   struct Action {
10     bytes32 seed;
11     bool play;
12  }
13  struct State {
14     bytes32[2] previousSeeds;
15     uint256[2] score;
16     uint256 partialScore;
17     bool progressiveAction;
18     Action savedAction;
19     bool hasPendingAction;
```

```
20      uint256 nextPlayer;
21  }
22  function startGame(bytes32 gameId, bytes32 startSeed) public
        returns(bool)
23  function endGame(bytes32 gameId) public
24  function playMove(bytes32 gameId, Action memory gameAction) public
        returns(State memory newState)
25  function validateMove(State memory state, Action memory action,
        uint256 playerNumber)
26  function advanceState(State memory state, Action memory action,
        uint256 playerNumber, bytes32 rngSeed)
```

Per gestire il pagamento delle quote di partecipazione abbiamo riusato il contratto FreezeBank descritto precedentemente.
Per iniziare una partita i giocatori, dopo aver depositato la propria quota, si accordano su un identificativo univoco che rappresenta la partita ed entrambi chiamano la funzione *startGame* fornendo il proprio seed iniziale.
Nella variabile *gameRecords* di tipo *GameInfo* per ciascuna partita vengono memorizzate le informazioni relative ai due giocatori e lo stato di progressione del gioco creato.
Ad ogni identificativo di gioco è inoltre associato una struttura di tipo *State*: questo stato è simile a quello usato nel contratto *Pig* originario con l'aggiunta di alcuni campi (*hasPendingAction, nextPlayer, savedAction, previousSeeds*) analoghi a quelli usati nel contatto *State Machine*.
Per aggiornare lo stato di gioco un giocatore invoca, attraverso una transazione, la funzione *playMove,* includendo l'azione proposta e il proprio seed.
L'avversario attraverso un'altra transazione chiama in modo analogo la funzione *playMove* fornendo il proprio seed per completare la generazione del numero casuale.
A questo punto il contratto chiama internamente la funzione *advance-*

State che aggiorna lo stato di gioco salvato nella blockchain.
Quando uno dei due giocatori raggiunge il punteggio vincente, viene aggiornata la variabile *gameStatus* e non è possibile proporre nuove azioni.
Uno dei due giocatori infine richiama la funzione *endGame* e le quote di partecipazione vengono ridistribuite in base al risultato della partita.
A differenza della nostra soluzione, per giocare una partita non è necessario firmare alcun dato e non è richiesto alcun canale di comunicazione diretto tra i due giocatori.

5.3.2 Costi relativi a PigOnChain

Come per le stime precedenti il costo del gas è stato fissato a 20 *gwei* per tutte le prove.
Per il deploy del contratto è necessario impiegare *3027843 gas*, per un costo di 0.06055686 *ether* equivalenti a circa 9.39 €.
Nella tabella 5.3 sono illustrati i costi delle funzioni del contatto, calcolati facendo la media tra alcune esecuzioni di prova del contratto.

Funzione	Gas	Costo ETH	Costo €
createGame()	171595	0.0034319	0.53
playMove()	125518	0.00251036	0.39
endGame()	46600	0.000932	0.14

Tabella 5.3: Costo delle funzioni del contratto PigOnChain.

5.3.3 Stima del costo per una partita

Stimare la durata di una partita di Pig non è facile, essendo un gioco fortemente influenzato dalle decisioni dei giocatori e dall'alea.
La letteratura relativa a strategie di gioco ottimali per questo gioco è molto vasta, ma per il nostro scopo abbiamo scelto la strategia più semplice descritta: un giocatore continua a tirare il dado fino a che non

raggiunge un punteggio parziale di almeno 20 [76].
Simulando molte partite basandoci su questa strategia abbiamo determinato che in media sono necessari 73 lanci di dado distribuiti tra i due giocatori.
Basandoci su questo dato è possibile stimare il costo di una partita svolto interamente su blockchain, descritto nella tabella 5.4.

Funzione	Numero Chiamate	Costo ETH	Costo €
createGame()	2	0.0068638	1.06
playMove()	146	0.36651256	56.81
endGame()	1	0.000932	0.14
	Totale	0.37430836	58.01

Tabella 5.4: Costo di una partita on chain di 73 mosse.

Appare evidente questi costi non siano convenienti per una singola partita e che quindi per giochi con molti turni come nel nostro caso basarsi solamente sulla blockchain non è fattibile.
Inoltre per ogni mossa occorrerebbe attendere la validazione di almeno 2 blocchi, considerando un tempo medio di 15 secondi a blocco, un'intera partita avrebbe un *downtime* di circa 37 minuti!
Con la nostra soluzione basata su state channel, nel caso ideale dove i due giocatori non discordano mai per aggiornare lo stato di gioco, una partita richiede solamente due transazioni, una per aprire il canale e una per chiuderlo.
Sommando i costi presenti nella tabella 5.2, una partita costerebbe in totale solamente 0.00674716 *ether* equivalenti a circa 1.05€.
Inoltre i tempi di latenza tra una mossa e l'altra sono trascurabili, grazie a WebRTC è possibile scambiarsi messaggi in tempo reale, comportando un'esperienza di gioco sicuramente migliore.
Le uniche volte che è necessario attendere la validazione di una transazione su blockchain sono in occasione dell'apertura e la chiusura dello state channel e nel caso non si concordi sull'aggiornamento dello stato

con il proprio avversario.
Ogni disputa sullo stato tra i due giocatori ha un costo di 0.00814684 *ether* equivalenti a circa 1.26€.
Con un numero alto di dispute aperte anche il costo di una partita attraverso gli state channel può diventare poco conveniente, ma introducendo un sistema di penalità adeguato è facile incentivare i giocatori a comportarsi onestamente.
Inoltre, introducendo la possibilità di chiudere il canale in seguito ad una disputa, i giocatori possono decidere di smettere di giocare se valutano che continuare a giocare non è più conveniente per loro.

Conclusioni

L'applicazione sviluppata per questa tesi soddisfa gli obiettivi che ci eravamo posti: è una soluzione modulare che consente di implementare nuovi giochi per due giocatori, anche più complessi di quello scelto nel nostro esempio, riutilizzando gran parte del codice.
Sfruttando la possibilità di firmare messaggi attraverso il wallet Ethereum è stato possibile raggiungere un'esperienza di gioco soddisfacente, senza la necessità di pagare commissioni o attendere i tempi di validazione per ogni transazione.
Nel caso migliore nel quale i giocatori non si trovano mai in disaccordo sull'avanzamento dello stato di gioco, per completare una partita sono necessarie solamente le due transazioni usate per aprire e chiudere lo state channel.
Tutte le azioni di gioco che modificano lo stato di gioco sono calcolate *off chain* senza quindi incorrere nelle limitazioni dovute ai problemi di scalabilità della blockchain.
Con un'adeguata posta in gioco e delle forti penalità per i giocatori che violano il protocollo di gioco, è possibile scoraggiare il più possibile i giocatori a comportarsi in modo disonesto limitando quindi al minimo le interazioni con la blockchain.
La soluzione trovata per la connessione tra i client di gioco è pratica, anche se sarebbe interessante per un futuro sviluppo rimuovere la necessità del server centralizzato, usato per la fase iniziale del protocollo di connessione, attraverso altre soluzioni peer-to-peer.

I risultati degli esperimenti sono abbastanza soddisfacenti, il costo delle transazioni è abbastanza contenuto e le tempistiche misurate dalle transazioni effettuate *on chain* rientrano nell'intervallo di tempo minimo richiesto per la validazione di un blocco.
Secondo i test effettuati il totale del costo per l'apertura e la chiusura di uno state channel è poco più di un euro e risolvere le dispute ha costi leggermente più alti ma abbastanza equivalenti.
Testando uno smart contract che permette di giocare allo stesso gioco scelto interamente su blockchain ha rivelato costi invece impraticabili, dimostrando l'utilità della nostra soluzione.
Le dimensioni del contratto relativo allo state channel si avvicinano pericolosamente al limite massimo di codice che è possibile inserire in un singolo contratto, limitandone eventuali espansioni future.
La ridondanza inserita all'interno delle strutture dati usate per ragioni di sicurezza potrebbe essere in parte superflua, cambiando alcuni elementi del protocollo: sarebbe interessante provare a semplificare alcune parti della implementazione dei contratti, rinunciando a parte della modularità su cui si basa la suddivisione netta tra i vari smart contract implementati.
Un'altra ulteriore frontiera di sviluppo per il futuro potrebbe essere generalizzare il protocollo per un numero di giocatori superiore a due, questo però comporterebbe rivoluzionare tutto il meccanismo delle dispute previsto nell'attuale architettura del protocollo.

Bibliografia

[1] L. Law, S. Sabett, and J. Solinas, "How to make a mint: the cryptography of anonymous electronic cash," *Am. UL Rev.*, vol. 46, p. 1131, 1996.

[2] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual International Cryptology Conference*. Springer, 1992, pp. 139–147.

[3] W. Dai, "B-money-an anonymous, distributed electronic cash system," 1998.

[4] H. Finney, "Rpow: Reusable proofs of work," *Internet: https://cryptome. org/rpow. htm*, 2004.

[5] N. Szabo, "Bit gold," *Website/Blog*, 2008.

[6] F. Tschorsch and B. Scheuermann, "Bitcoin and beyond: A technical survey on decentralized digital currencies," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016.

[7] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[8] J. Lansky, "Possible state approaches to cryptocurrencies," *Journal of Systems Integration*, vol. 9, no. 1, pp. 19–31, 2018.

[9] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc.", 2014.

[10] A. Back *et al.*, "Hashcash-a denial of service counter-measure," 2002.

[11] D. Eastlake and T. Hansen, "Us secure hash algorithms (sha and sha-based hmac and hkdf)," RFC 6234, May, Tech. Rep., 2011.

[12] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[13] V. Buterin *et al.*, "Ethereum white paper: a next generation smart contract & decentralized application platform," *First version*, 2014.

[14] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[15] Ethereum, "ethereum/solidity," Sep 2019. [Online]. Available: https://github.com/ethereum/solidity

[16] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *Ieee Access*, vol. 4, pp. 2292–2303, 2016.

[17] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, p. 2, 1996.

[18] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'Reilly Media, 2018.

[19] “What are smart-contracts and decentralized applications?” [Online]. Available: https://docs.ethhub.io/ethereum-basics/what-is-ethereum/#what-are-smart-contracts-and-decentralized-applications

[20] E. Foundation, “Merkling in ethereum.” [Online]. Available: https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/

[21] “Gas costs from yellow paper – eip-150 revision (1e18248 - 2017-04-12).” [Online]. Available: https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWcOlRem_mO09GtSKEKrAsfO7Frgx18pNU/edit#gid=0

[22] “Introduction to smart contracts.” [Online]. Available: https://solidity.readthedocs.io/en/v0.5.11/introduction-to-smart-contracts.html#transactions

[23] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer *et al.*, “On scaling decentralized blockchains,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 106–125.

[24] J. Stark, “Making sense of ethereum’s layer 2 scaling solutions: State channels, plasma, and truebit,” Feb 2018. [Online]. Available: https://medium.com/l4-media/making-sense-of-ethereums-layer-2-scaling-solutions-state-channels-plasma-and-truebit-22cb40dcc2f4

[25] S. Raval, *Decentralized applications: harnessing Bitcoin’s blockchain technology*. " O’Reilly Media, Inc.", 2016.

[26] O. Kharif, “Cryptokitties mania overwhelms ethereum network’s processing,” *Bloomberg (4 Dec. 2017)*.

[27] V. Buterin, “Sharding faq.” [Online]. Available: https://github.com/ethereum/wiki/wiki/Sharding-FAQ

[28] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, “On the security and performance of proof of work blockchains,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 3–16.

[29] T. McConaghy, “The dcs triangle.” [Online]. Available: https://blog.bigchaindb.com/the-dcs-triangle-5ce0e9e0f1dc

[30] G. Slepak and A. Petrova, “The DCS theorem,” *CoRR*, vol. abs/1801.04335, 2018. [Online]. Available: http://arxiv.org/abs/1801.04335

[31] “Ethereum 2.0 specifications.” [Online]. Available: https://github.com/ethereum/eth2.0-specs

[32] F. Saleh, “Blockchain without waste: Proof-of-stake,” *Available at SSRN 3183935*, 2019.

[33] “Proof of stake (pos).” [Online]. Available: https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake/

[34] I.-C. Lin and T.-C. Liao, “A survey of blockchain security issues and challenges.” *IJ Network Security*, vol. 19, no. 5, pp. 653–659, 2017.

[35] I. Eyal and E. G. Sirer, “Majority is not enough: Bitcoin mining is vulnerable,” *Communications of the ACM*, vol. 61, no. 7, pp. 95–102, 2018.

[36] A. Akentiev, “Nothing-at-stake and longrange-attack in pos,” Sep 2017. [Online]. Available: https://blog.goldmint.io/nothing-at-stake-and-longrange-attack-in-pos-4ec486f1fc89

[37] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *CoRR*, vol. abs/1710.09437, 2017. [Online]. Available: http://arxiv.org/abs/1710.09437

[38] "'phase zero' of a new ethereum blockchain could go live next january." [Online]. Available: https://www.coindesk.com/phase-zero-of-a-new-ethereum-blockchain-could-go-live-next-january

[39] "Ethereum sharding explained." [Online]. Available: https://education.district0x.io/general-topics/understanding-ethereum/ethereum-sharding-explained/

[40] "Sharding." [Online]. Available: https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/sharding/

[41] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," *URL: http://www.opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, p. 72, 2014.

[42] G. Chaumont, "We are switching to delegated proof-of-stake," Jun 2019. [Online]. Available: https://medium.com/x-cash/we-are-switching-to-delegated-proof-of-stake-5b3b0ac14d0d

[43] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, pp. 1–47, 2017.

[44] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.

[45] J. Coleman, L. Horne, and L. Xuanji, "Counterfactual: Generalized state channels," 2018.

[46] P. McCorry, S. Bakshi, I. Bentov, A. Miller, and S. Meiklejohn, “Pisa: Arbitration outsourcing for state channels.” *IACR Cryptology ePrint Archive*, vol. 2018, p. 582, 2018.

[47] P. McCorry, C. Buckland, S. Bakshi, K. Wüst, and A. Miller, “You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies,” 2018.

[48] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, “Multi-party virtual state channels,” in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 625–656.

[49] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. Leung, “Decentralized applications: The blockchain-empowered software system,” *IEEE Access*, vol. 6, pp. 53 019–53 033, 2018.

[50] BesanciaHime, “"play to earn": New rules, new players,” Jul 2019. [Online]. Available: https://nonfungible.com/blog/play-to-earn-new-rules-new-players

[51] “Blockchain gaming part ii: Successes and failures of first-generation games,” Aug 2019. [Online]. Available: https://www.theblockcrypto.com/post/35251/blockchain-gaming-part-ii-successes-and-failures-of-first-generation-games

[52] Xaya, “Just how much money have people made with huntercoin?” Mar 2019. [Online]. Available: https://medium.com/@XAYA/just-how-much-money-have-people-made-with-huntercoin-7c7336f5ac04

[53] CryptoKitties, “The kittyverse: Community-built experiences for your cryptokitties using the nifty™ license.” [Online]. Available: https://www.cryptokitties.co/kittyverse

[54] J. Nielsen, *Usability engineering*. Elsevier, 1994.

[55] “How long does an ethereum transaction really take?” Sep 2019. [Online]. Available: https://ethgasstation.info/blog/ethereum-transaction-how-long/

[56] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, “Probabilistic smart contracts: Secure randomness on the blockchain,” *arXiv preprint arXiv:1902.07986*, 2019.

[57] J. Longley, “Funfair technology roadmap and discussion,” *WhitePaper*. [Online]. Available: https://funfair.io/wp-content/uploads/FunFair-Technical-White-Paper.pdf

[58] J. Scarne and C. Rawson, *Scarne on dice*. Crown, 1980.

[59] T. W. Neller, C. G. Presser, I. Russell, and Z. Markov, “Pedagogical possibilities for the dice game pig,” *Journal of Computing Sciences in Colleges*, vol. 21, no. 6, pp. 149–161, 2006.

[60] D. Flanagan, *JavaScript: the definitive guide*. " O’Reilly Media, Inc.", 2006.

[61] “Metamask ethereum browser extension.” [Online]. Available: https://metamask.io/

[62] “Ethereum api: Ipfs api & gateway: Eth nodes as a service.” [Online]. Available: https://infura.io/

[63] “Components and props.” [Online]. Available: https://reactjs.org/docs/components-and-props.html

[64] A. Boduch, *Flux architecture*. Packt Publishing Ltd, 2016.

[65] C. Gackenheimer, “Introducing flux: An application architecture for react,” in *Introduction to React*. Springer, 2015, pp. 87–106.

[66] S. P. R. Salvatore Loreto, *Real-Time Communication with WebRTC: Peer-to-Peer in the Browser*, 1st ed. O’Reilly Media, 2014.

[67] “A study of webrtc security.” [Online]. Available: https://webrtc-security.github.io/

[68] R. Bloemen, L. Logvinov, and J. Evans, “Github,” Sep 2017. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md

[69] Ethereum, “Eip170.” [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md

[70] S. K. Kumar, “About abi encoder v2.” [Online]. Available: https://ethereum.stackexchange.com/questions/64562/about-abi-encoder-v2

[71] M. Wohrer and U. Zdun, “Smart contracts: security patterns in the ethereum ecosystem and solidity,” in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[72] “Top 50 cryptocurrency prices.” [Online]. Available: https://www.coinbase.com/price

[73] T. H. Hale, “Comparison of the different testnets,” Jan 1968. [Online]. Available: https://ethereum.stackexchange.com/questions/27048/comparison-of-the-different-testnets

[74] Ethereum, “ethereum/ropsten.” [Online]. Available: https://github.com/ethereum/ropsten/blob/master/revival.md

[75] “Eth gas station.” [Online]. Available: https://ethgasstation.info/

[76] T. W. Neller and C. G. Presser, “Practical play of the dice game pig,” *The UMAP Journal*, vol. 31, no. 1, 2010.

Ringraziamenti

Finalmente anche quest’ultimo mio percorso di studi è giunto a termine e mi sembra doveroso ringraziare tutte le persone che mi sono state accanto in questo periodo trascorso a Bologna.
Innanzitutto devo ringraziare la mia famiglia per aver reso possibile questa mia esperienza fuori sede e per avermi sempre sostenuto ed incoraggiato.
In questi tre anni a Bologna ho conosciuto tante persone fantastiche che ringrazio per i momenti memorabili che abbiamo passato insieme e per essermi state accanto nei momenti di difficoltà.
Ovviamente ringrazio anche tutti i miei amici di più lunga data con cui sono rimasto sempre in contatto nonostante la lontananza.
Infine ringrazio il prof. Stefano Ferretti, relatore di questa tesi di laurea, per la disponibilità ed i suggerimenti che mi ha offerto durante questi mesi di studio.